AD-A082 216   GENERAL RESEARCH CORP SANTA BARBARA CA SYSTEMS TECHNO--ETC   F/G 9/2

ADAPTIVE SEARCH TECHNIQUES APPLIED TO SOFTWARE TESTING.(U)

FEB 80   J P BENSON, D M ANDREWS   F49620-79-C-0115

UNCLASSIFIED   GRC-CR-1-925   AFOSR-TR-80-0234   NL

1 OF 1

AD A082216

END

DATE FILMED

4-80

DTIC

18 19

AFOSR-TR- 80 - 0234

LEVEL 12

14

GRC-CR-1-925

6

Adaptive Search Techniques Applied to Software Testing

9

Final Report

Jeoffrey

by

10

J.P. Benson

D.M. Andrews

Dorothy

11

February 1980

12 92

16 2304

17 A2

AD A082216

SYSTEMS TECHNOLOGIES GROUP

GENERAL RESEARCH CORPORATION

A SUBSIDIARY OF FLOW GENERAL INC.

P.O. Box 6770, Santa Barbara, California 93111

Sponsored by
Air Force Office of Scientific Research
Bolling Air Force Base
Washington, D.C.

15

Under Contract F49620-79-C-0115

411 275

Approved for public release; distribution unlimited.

80 3 20 080

In addition to approval by the Project Leader and Department Head, General Research Corporation reports are subject to independent review by a staff member not connected with the project. This report was reviewed by R. L. Stone.

Sponsored by

Air Force Office of Scientific Research
Under Contract F49620-79-C-0115

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 80-0234	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ADAPTIVE SEARCH TECHNIQUES APPLIED TO SOFTWARE TESTING		5. TYPE OF REPORT & PERIOD COVERED Final
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) J. P. Benson, D. M Andrews		8. CONTRACT OR GRANT NUMBER(s) F49620-79-C-0115
9. PERFORMING ORGANIZATION NAME AND ADDRESS General Research Corp. P. O. Box 6770 Santa Barabra, CA 93111		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Research/NM Bolling AFB, Washington, DC 20332		12. REPORT DATE February 1980
		13. NUMBER OF PAGES 88
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

An experiment was performed in which executable assertions were used in conjunction with search techniques in order to test a computer program automatically. The program chosen for the experiment computes a position on an orbit from the description of the orbit and the desired point.
Errors were inserted in the program randomly using an error generation method based on published data defining common error types. Assertions were written for the program and it was tested using two different techniques. The first divided up the range of the input variables and selected test cases from within

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

20. Abstract cont.

the subranges. In this way a "grid" of test values was constructed over the program's input space.

The second used a search algorithm from optimization theory. This entailed using the assertions to define an error function and then maximizing its value. The program was then tested by varying only a limited number of the input variables and a second time by varying all of them. The results indicate that this search testing technique was as effective as the grid testing technique in locating errors and was more efficient. In addition, the search testing technique located critical input values which helped in writing correct assertions.

Abstract

An experiment was performed in which executable assertions were used in conjunction with search techniques in order to test a computer program automatically. The program chosen for the experiment computes a position on an orbit from the description of the orbit and the desired point.

Errors were inserted in the program randomly using an error generation method based on published data defining common error types. Assertions were written for the program and it was tested using two different techniques. The first divided up the range of the input variables and selected test cases from within the subranges. In this way a "grid" of test values was constructed over the program's input space.

The second used a search algorithm from optimization theory. This entailed using the assertions to define an error function and then maximizing its value. The program was then tested by varying only a limited number of the input variables and a second time by varying all of them. The results indicate that this search testing technique was as effective as the grid testing technique in locating errors and was more efficient. In addition, the search testing technique located critical input values which helped in writing correct assertions.

CONTENTS

CONTENTS (cont.)

FIGURES

TABLES

Acknowledgement

This research project was sponsored by the Air Force Office of Scientific Research, Contract Number F49620-79-C-0115. Lt. Col. George W. McKemie was the program manager.

Participating in the project were J. P. Benson, principal investigator, D. M. Andrews, N. B. Brooks, R. N. Meeson, and D. W. Cooper.

1 INTRODUCTION

Although Dijkstra's famous comment on testing, that it will never show the absence of bugs, only their presence, is undoubtedly true, testing is still the method most used for showing the correctness of software. if testing is to be used, ways must be found to make it more efficient and effective.

A paper by Alberts[1] presents data indicating that testing and validation efforts account for approximately 50% of the cost of developing a software system, where development includes the typical phases of conceptual design, requirements analysis, development, and operational use. This cost includes those associated with locating the errors, correcting the errors (which may include redesign), and checking that the corrections have removed the cause of the error. The testing process is a very labor-intensive activity, as is any aspect of software development. If methods could be found to automate the testing process, the cost of developing software could be reduced.

1.1 PROBLEMS WITH TESTING

Two of the many problems involved in testing software are (1) how to develop test cases which identify errors and (2) how to check the results from these test cases. Before software testing can be automated and its cost reduced, these two problems must be solved.

Many methods have been proposed for identifying test cases which will show that a program performs correctly or indicate the errors which are present in the program. For examples of these methods see Howden[2]

---

[1] D. S. Alberts, "The Economics of Software Quality Assurance" in AFIPS Conference Proceedings: 1976 National Computer Conference, Vol. 45, AFIPS Press, Montvale, N.J. pp. 433-442.

[2] W. E. Howden, "Theoretical and Empirical Studies in Program Testing," IEEE Transactions on Software Engineering, Vol. SE-4, July 1978.

and Gannon[1]. Basically, the problem is one of complexity. For most programs, the number of different combinations of input values is practically infinite. Therefore, using exhaustive testing to show that a program works correctly is an impossible task.

Given the fact that programs cannot be tested by trying all test cases, what are the alternatives? Boundary value testing, path testing, and symbolic execution[2] have been some of the suggested solutions. The key problem is finding test cases which detect the errors present in the software. At present, there are no methods for deriving test cases with this property although many studies of the types of errors commonly found in software have been undertaken.[3-5]

The second problem has to do with checking whether a test has been successful. Even if there were a method for selecting test cases which was able to identify specific errors in a program, the process of evaluating whether or not the program ran successfully is a manual one. The output from the program must be compared with the expected results. For large programs composed of many functions this is a very time-consuming task.

1.2 A PROPOSED SOLUTION

From the above discussion, it is evident that automating the testing of computer programs requires finding methods for developing

---

[1] C. Gannon, "Error Detection Using Path Testing and Static Analysis," Computer, Vol. 12, August 1979.

[2] L. A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," IEEE Transactions on Software Engineering, Vol. SE-2, September 1976.

[3] T. A. Thayer et al., Software Reliability Study, TRW Defense and Space Systems Group, RADC-TR-76-238, Redondo Beach, Calif., August 1976.

[4] M. J. Fries, Software Error Data Acquisition, Boeing Aerospace Company, RADC-TR-77-130, Seattle, Washington, April 1977.

[5] Verification and Validation for Terminal Defense Program Software: The Development of a Software Error Theory to Classify and Detect Software Errors, Logicon HR-74012, May 1974.

effective test cases as well as methods for efficiently evaluating the results of using them. A method for solving these problems has been developed that combines the use of search algorithms from operations research with executable assertions from software verification research.

Finding the maximum or minimum value of a function of several variables each subject to some set of constraints is a common problem in operations research. Minimizing the cost of constructing a building given the choice of using brick, wood, and adobe materials in different proportions typifies problems of this sort. Many methods have been developed for solving such problems, for example, see Denn.[1] One of the simplest is to define the parameter of interest (e.g., cost) as a function of the possible alternatives (e.g. brick, wood, adobe). The problem then is to find a minimum value of the function defined by the values of the alternatives (variables). Figure 1.1 illustrates this for two variables, brick and wood. The cost function defines a surface, with "hills" (maximums) and "valleys" (minimums).

The goal is to find a point on this surface which is a minimum (in the example of building cost). This point corresponds to a particular set of values of the alternatives or variables. Finding such a minimum value requires that this surface be searched. There are many methods for traversing the surface according to some search heuristic (for example, in the direction of the gradient) until a solution is found.

The problem of evaluating the results limits the application of these techniques to the testing of computer programs. That is, in operations research, we are usually trying to maximize or minimize the

---

[1] M. M. Denn, Optimization by Variational Methods, New York, McGraw-Hill, 1969.

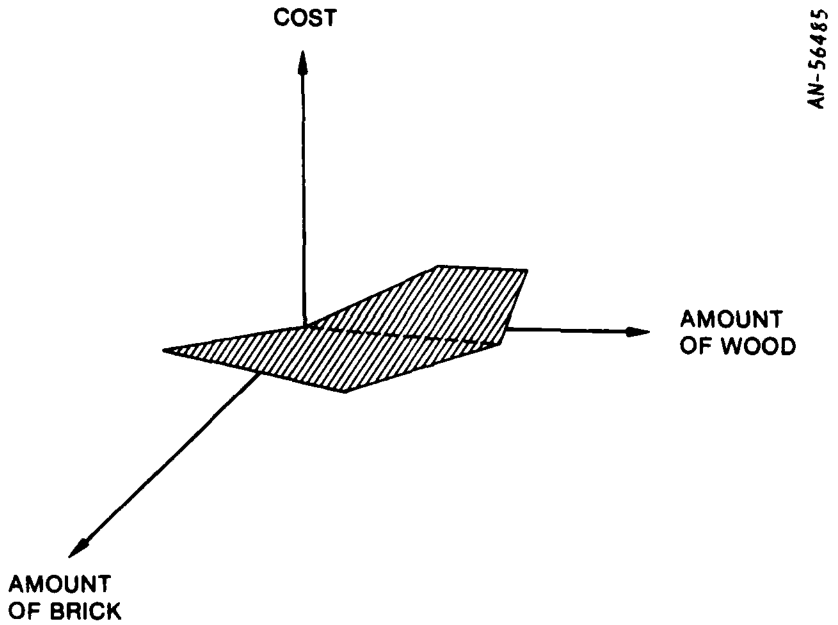

Figure 1.1. Cost as a Function of Building Material

value of one variable, whereas in software testing we are usually trying to compare the value of many output variables with their expected values.

The solution to this problem has been found in "executable assertions," a technique developed for proving software correct and for checking it while it is running. Assertions are comments added to a program which specify how the program is to behave. They may specify a range of values for a variable, the relation the values of two or more variables have to each other or compare the state of a present computation to that of a past computation. Figure 1.2 shows an example of two assertions that specifies the range of values that the variable VALUE can assume.

To make an assertion "executable," we merely translate it into machine language. Then while the program is running, the assertion can

ASSERT (VALUE .GE. 0.0)

ASSERT (VALUE .LE. TWOPI)

Figure 1.2. Examples of Assertions

be evaluated. As in the case of a logical function, the assertion has a value of true or false. If the value of an assertion becomes false at any point in the execution of a program, then this can be reported as any other error message.

1.3 COMBINING ASSERTIONS AND SEARCH ALGORITHMS

Assertions give us a method for evaluating whether a program has run correctly without looking at all of its output. If the assertions are written correctly and they completely specify the algorithm, then the correctness of the program can be determined while the program is running. This is not to say that writing assertions to accomplish this is easy; a comprehensive and complete set of assertions for a program is difficult to develop. But if it can be done, then the problem of examining the output of a program to determine whether it executed a test case correctly has been solved.

Since using assertions means that we no longer have to examine the output of a program, the automated testing of computer programs becomes possible--provided we can automate the selection of test cases. If we can transform the output from the assertions into a function, we can utilize the search techniques from operations research to locate errors.

The basic idea is this: The function we define is the number of assertions that become false during the execution of a particular test case. The independent variables are the values of the input variables of the program. The search technique will be used to find the values of

the input variables for which the maximum number of assertions are violated. The function relating the number of assertions violated to the values of the input variables is called the "error function," and the surface that it describes is called the "error space."

If the search algorithm is to perform correctly, the error function must (1) not define a flat (uniform) surface and (2) not be discontinuous (have spikes) at any points. A previous experiment,[1] investigated the error function for a scheduling program. It was found that the error function for this program was neither uniform nor discontinuous. In a second experiment, described below, we have attempted to show that this is also true for another program "seeded" with several types of errors. We have also attempted to determine the efficiency of the search technique in locating these errors relative to other types of testing methods.

1.4 OVERVIEW OF THE EXPERIMENT

The experiment was to select a program, add assertions to it, and seed it with errors from a list of typical software errors. The location of the errors was determined randomly. Each of the errors was inserted in the program one at a time and the program was then tested by systematically choosing combinations of values for the input parameters. This testing was done automatically by a program which varied the input parameters over the required values. After this, the program was tested by the search routine, first by allowing the search algorithm to vary the same variables that were varied in the first tests, and then allowing it to vary all of the input variables.

1.5 THE PROGRAM

The program selected takes an orbit described by six independent parameters (longitude of the ascending node, inclination of the orbit

[1] J. Benson, A Preliminary Experiment in Automated Software Testing, General Research Corporation TM-2308, February 1980.

plane, angle of the perigee, eccentricity, time at perigee, and semi-major axis) and converts this description into a state vector representation of a point on the orbit (time, postion, velocity, and acceleration). The point is determined by the values of two other parameters. The range of values of one of these parameters is dependent upon the other. In all, there are ten input parameters, seven of which are independent of the others.

1.6 THE SEARCH ROUTINE

The search routine chosen for the experiment was one developed by Box[1] called complex search. This algorithm constructs a hypertriangle, or complex, of the values of the function from several tests and then rotates, shrinks, expands, and projects the complex in order to locate a value which is larger (in the case of finding the maximum) than the worst point currently in the complex. The worst point is then replaced by the new point and the process continued until no further progress can be made.

1.7 THE TEST DRIVER

Several programs were also written in order to support the testing and make it as automatic as possible: (1) A test driver, which handled the selection of the testing method to be used and read in an initial test case was written, (2) a set of subroutines which implemented the constraints among the input variables used in generating new values for the search routine, and (3) a set of routines to count the number of assertions violated in each test and print the results.

1.8 THE ASSERTIONS

Assertions added to the program were of three types: (1) those that described ranges of variable values, (2) those that described the relationship between values of variables, and (3) those which kept track of the history of the computation. Two routines were also written which included assertions to check the values of the input variables and the

---

[1] M. J. Box, "A New Method of Constrained Optimization and Comparison with Other Methods," Computer Journal, Vol. 8, 1965.

correctness of the results. These routines were invoked at the beginning of the test program and at the end of the test program.

1.9 SELECTING ERRORS

Certain categories of errors were selected from a list of common software errors. Errors of these types were inserted into the test program by randomly selecting sites (statements in the program) where the particular type of error could occur.

1.10 TESTING TECHNIQUES

The program was then tested by inserting one error at a time. First, the program was tested by taking combinations of values from three input variables. The permissible input range of each of the variables was divided up into equal subranges so that a reasonable number of test cases could be performed. Test values for each variable were selected by choosing the end-points of each subrange. The program was then tested using the selected values for the three input variables. First, the values of two of the three variables were fixed at a value selected from their range of test values. Then, a test was run for each of the test values of the third variable. The value of the third variable was then fixed, and the first variable was varied over its set of test values. After this, the values of the first and third variable were fixed and the second variable was varied. The testing continued until all combinations of the test values for the three varibles had been used. In this way a "grid" over the input space was obtained. The values of the variables which caused assertions to be violated and the number of assertions violated were recorded.

A majority of the errors (15 out of 24) were not detected by the original assertions for a number of reasons. Two of the errors were not detected since they occurred only if another error had occurred previously during program execution. For other errors, it was found to be very difficult to write assertions that would detect them. Finally,

eight of the errors were not detected simply because the program did not contain enough assertions. In order to investigate the performance of the search algorithm, new assertions were added to the program and the grid tests were run again. Errors which were not detected in this second set of tests were removed from the list of errors used in the experiment.

Next, the errors were again inserted one at a time and the search routine was allowed to vary only the variables which were varied in the grid tests. The number of assertions violated and the input values which caused the violations were recorded.

Finally, the errors were again used one at a time; but this time the search routine was allowed to vary any of the seven independent variables in order to locate a maximum. Again, the assertions violated and the input values which caused the violations were recorded.

1.11 RESULTS

The results from the grid tests demonstrated the effectiveness of the assertions in detecting the errors. Table 1.1 shows the results of these tests. Of the original 24 errors, nine (thirty-eight percent) were detected by the original assertions, and eight (thirty-three percent) were detected by the assertions that were added. (The seven errors, twenty-nine percent, which could not be detected by assertions, were not tested).

The relative effectiveness of the search testing methods versus the grid testing method is summarized in Table 1.2. (In this table, and those following, the "error number" column refers to a unique number assigned to each error by the error generation method discussed in Sec. 4.) In one case, the grid technique caused an assertion violation which

TABLE 1.1

RESULTS FROM GRID TESTS

	Number	Percentage
Errors Detected by Original Assertions	9	38
Errors Detected by Added Assertions	8	33
Errors Not Detected by Assertions	7	29
Total	24	100

TABLE 1.2

EFFECTIVENESS OF SECOND TESTING TECHNIQUES

	Number of Assertion Violations Detected by Testing Technique		
Error Number	Grid	3-Variable Search	All-Variable Search
14	1	2	3
28	1	0	0
47	2	2	1
74	7	7	8

neither search technique caused. In another case, the search technique using all variables was not able to cause an assertion violation that was caused by the grid technique and the search varying three variables. On the other hand, the search technique using all variables was able to cause an assertion violation which neither the grid technique nor the search using three variables was able to cause. Finally, in one case the search technique using three variables caused an assertion violation that the grid technique did not cause while the search using all variables caused another assertion violation in addition to the one discovered by the search using three variables. In all other tests, each of the methods caused the same assertions to be violated.

The efficiency of the search technique was not measured directly, but an estimate of the behavior of the all-variable search technique in relation to the grid technique can be given. Except for error 52, which required 683 tests, the grid technique required 317 tests. In the case of the search method which varied all input variables, Table 1.3 shows, for each error, the number of the test in which the first assertion violation was detected. In all, fifteen of the seventeen detectable errors were detected by the seventh test in the search.

TABLE 1.3

DETECTION OF ASSERTION VIOLATIONS BY SEARCH METHOD

Error Number	Test Number of First Assertion Violation
1	5
3	2
13	7
14	5
28	*
31	4
37	5
41	3
47	57
48	3
52	3
54	3
56	5
57	7
64	2
67	5
74	2

*No assertion violations detected.

2 THE TEST PROGRAM

The program selected for testing is one of a number of subroutines in a program library called TRAID.[1] This set of programs is used to compute solutions to orbital mechanics problems. The particular program, ORBP, was written in 1968 and has been used extensively since that time. It has undergone several revisions. The function of the program is to take as input an orbit described by a set of eight parameters or orbital elements (only six of which are independent), and produce from this set a state vector representation of a point on the orbit. The state vector includes the time, position, velocity and acceleration in three dimensions. The particular point on the orbit is specified by a parameter (MODE), which, in conjunction with another parameter (VALUE), allows the state vector describing the point to be computed. (For a simple discussion of the methods for describing orbits see Macko.[2]) The orbital element vector is shown in Table 2.1 along

TABLE 2.1

ORBITAL ELEMENT VECTOR PARAMETERS

	Parameter	Range
1.	Longitude of the ascending node	0 to 2π
2.	Inclination of the orbit plane	0 to π
3.	Angle of the perigee	0 to 2π
4.	Semi-latus rectum	dependent
5.	Eccentricity (E)	0.1 to 0.9
6.	Time at perigee	0 to period
7.	Period divided by 2π	dependent
8.	Semi-major axis (A)	6,375,180 to 35,861,000 meters

[1] T. Plambeck, The Compleat Traidsman, General Research Corporation IM-711/2, revised edition, September 1969.

[2] S. J. Macko, Satellite Tracking, John F. Rider Publisher, Inc., New York, 1962.

with the ranges of each independent parameter as used to test the program. The letters E and A are used to indicate the eccentricity and semi-major axis respectively.

An orbit is described by the following eight parameters: (1) longitude of the ascending node, (2) inclination of the orbit plane, (3) argument (angle) of the perigee, (4) semi-latus rectum, (5) eccentricity, (6) time at perigee, (7) period divided by two pi, and (8) the semi-major axis. Of these eight paramenters, the semi-latus rectum and the period are dependent upon the others; they are included in the vector only to simplify the calculations. The way in which these parameters are calculated from the others is shown in Fig. 2.1.

The output state vector is shown in Table 2.2. It includes the time at the point on the orbit, and the position, velocity and acceleration in three dimensions. These last parameters are given in a coordinate system relative to the center of the earth.

Semi-latus rectum = $A * (1 - E^2)$

where

A = semi-major axis

E = eccentricity

Period = $(A * \quad A/GCON) * 2\pi$

where

GCON = gravitational constant = 3.9857×10^{14}

Figure 2.1. Calculation of Dependent Orbital Parameters

TABLE 2.2

STATE VECTOR PARAMETERS

	Parameter
1.	Time
2.	X-coordinate
3.	Y-coordinate
4.	Z-coordinate
5.	X-velocity
6.	Y-velocity
7.	Z-velocity
8.	X-acceleration
9.	Y-acceleration
10.	Z-acceleration

Together, the parameters MODE and VALUE specify a point on the orbit. The possible values of the mode parameter and the corresponding ranges of the value parameter are shown in Table 2.3. The mode parameter directs ORBP to perform one of six possible computations to locate a point on the orbit specified by the orbital element vector. The MODE parameter indicates how the VALUE parameter is to be interpreted. That is, the value parameter is only a number, the MODE parameter indicates what that number stands for. For example, MODE could indicate the time at the desired point, and therefore VALUE could assume any value between 0 and the period of the orbit. The six possible modes are: (1) angle of the point from the perigee point, (2) radius in the increasing direction (i.e., toward apogee), (3) radius in the decreasing direction (toward perigee), (4) time, (5) altitude in the increasing direction, and (6) altitude in the decreasing direction.

According to the values of MODE and VALUE, ORBP calculates a state vector using the orbital element vector. The calculations for altitude and radius are performed using the same code. This is done by adding

TABLE 2.3

MODE AND VALUE PARAMETERS

Mode	Meaning of Value	Range of Value
0	Angle from perigee	0 to 2π
1	Increasing radius	R_{min} to R_{max}
2	Decreasing radius	R_{min} to R_{max}
3	Time	0 to Period
4	Increasing altitude	Alt_{min} to Alt_{max}
5	Decreasing altitude	Alt_{min} to Alt_{max}

---

the radius of the earth to VALUE if it corresponds to an altitude (MODE equal 4 or 5). (See Fig. 2.2.) The point on the orbit is found by computing the angle between the point and the perigee and the radius from the focus of the orbit (the center of the earth) to the point. The only loop in the program occurs when MODE indicates that VALUE is to be interpreted as time. In this case, an iterative algorithm is used to calculate the angle of the point from the perigee.

2.1 THE SEARCH ROUTINE

The search routine selected for the experiment was one invented by Box,[1] called complex search. It is a method for solving for the maximum or minimum of a nonlinear function. The independent values of the function may be limited by nonlinear inequality constraints. The independent values of the function along with the function value define a space. The set of values of the function define a hyperplane in the

---

[1] Box, op. cit.

Radius = A * (1 - E)

where

A = semi-latus rectum

E = eccentricity

Altitude = radius - RBODY

where

RBODY = radius of earth = 6,375,180 meters

Figure 2.2. Calculating Radius and Altitude

---

space. This hyperplane can then be expanded or contracted to find an extremum of the function. The hyperplane is called a "complex."

The technique is as follows. Choose a set of values of the independent variables at random (subject to constraints) and determine the value of the function from these values. The independent values and the function value define a point on the complex. Define other points in the same way until there is one more point than the number of independent variables in the function. Then replace the point with the worst function value with a new point. The new point is found by constructing the line formed by the rejected point and the centroid of the remaining points. A set of coefficients is then calculated to determine the exact location of the new point. These coefficients determine the degree of reflection, expansion, shrinkage, contraction, and rotation to be applied in forming the new set of points. New points are selected for the complex using the above technique until a solution is found. This technique is somewhat immune to irregularities (hills and valleys) in the surface being searched.

The search program was adapted from an implementation by Cooper which was used during the Adaptive Testing Experiment.[1] In general, the function may have many independent variables. In order for the search routine to function correctly, there must be one more point in the complex than there are independent variables in the function. For example, if the function has two independent variables, then the complex would have three vertices. That is, it would be a triangle. The coordinates of each vertex would be the values of the independent variables and the value of the function. For a function of two variables x and y, each complex point would have the coordinates $x_1, y_1, f(x_1, y_1)$ as shown in Fig. 2.3.

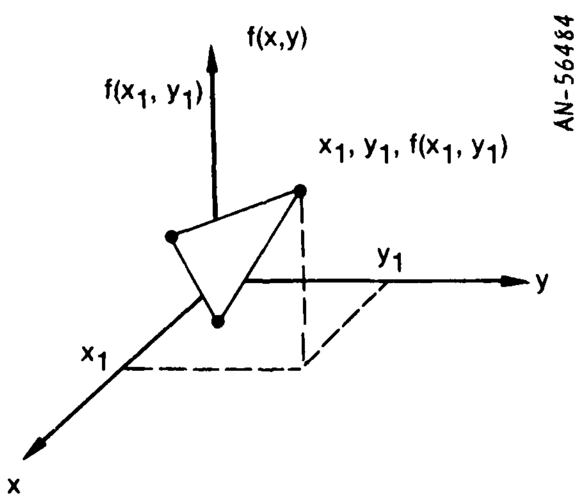

Figure 2.3. Coordinates of the Vertices of a Complex

[1] D. W. Cooper, Adaptive Learning Requirements and Critical Issues, General Research Corporation CR-4-708, January 1977.

The constraints were computed by a subroutine written especially for the test program. They included ranges of variables and relationships between the variables. Examples of the latter constraints include the relationship between the semi-major axis and the semi-latus rectum of the orbit (shown in Fig. 2.1) and the valid range of the VALUE parameter for different values of the MODE parameter (shown in Table 2.3).

An input parameter selects which independent variables are to be varied by the search algorithm. This was used in the experiment to vary only three of the independent variables in one test and all of the independent variables in the other test.

The termination condition for the search routine is determined by another input parameter. This parameter is a maximum function value which when found, reinitializes the search. When a set of input values has been found which causes this number of assertion violations, the maximum function value is increased by one and the search is begun again for this new value. If the new maximum value is not found, then the algorithm continues searching until one-hundred tests of the test program have been run.*

After constructing the complex, the search routine finds the worst point (minimum function value over all points in the complex) and tries to replace it with a point with a larger function value (assuming the maximum of the function is being sought). It does this by applying the operations of reflection, expansion, centroid substitution, contraction, shrinkage, and rotation to the complex in that order. In order to illustrate each of these operations, Fig. 2.4 shows the effect of each of these operations on a triangle. In "reflection" the new point is found by reflecting the old point through the centroid of the complex.

*Note that the "test number" column in Table 1.3 refers to the number of tests or runs of the test program (ORBP), not the search program. One run of the search program corresponds to at least 100 runs of the test program.

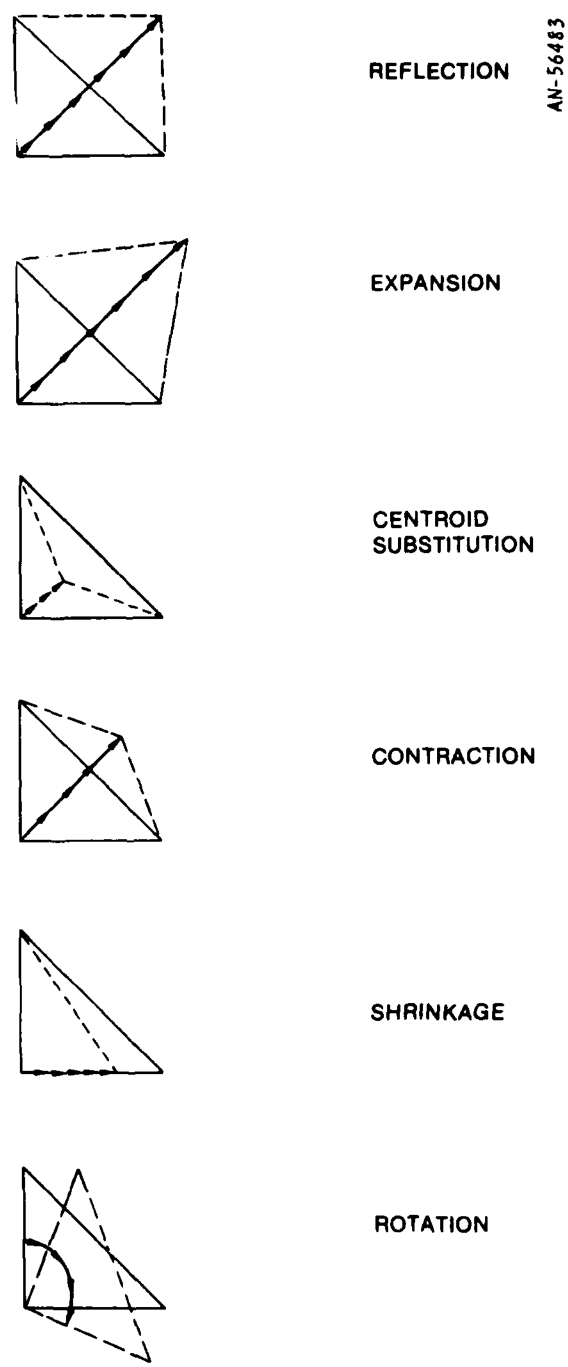

Figure 2.4. Complex Transformations

That is, the new point and the old point lie on a line through the centroid. The new point and the old point are each the same distance from the centroid. In "expansion," the distance of the new point from the centroid is greater than the distance from the old point to the centroid. In centroid substitution, the worst point is replaced by the centroid of the complex. "Contraction" reduces the distance from the new point to the centroid to be less than the distance from the old point to the centroid. "Shrinkage," instead of reflecting through the centroid of the complex, uses the point defined by the largest function value as the reflection point. Finally, "rotation" rotates the complex about the centroid in order to locate a new point. The cycle of operations continues until a maximum value is found or one-hundred tests have been run.

2.2 TEST DRIVER

A test driver was written to interface the search routine with the test program and initialize the test. The test driver determines which testing technique will be used: grid, search varying three variables, or search varying all variables. It initializes the values of all variables needed to conduct the test and reads in the basic set of orbital parameters which are common to all tests. It reads the values of the variables to be varied and their ranges and, for the grid test, divides the ranges up into intervals and selects a set of values for each variable corresponding to this division. It also calculates the dependent orbital paramenters (semi-latus rectum and period divided by 2π) and runs the grid tests. The search routine itself runs the search tests.

Other routines detect when assertions are violated, count the number of assertions violated in each of the tests, print a table of the assertions violated by test and record and print other information. The test program runs with other routines from the TRAID library which it uses to perform certain computations and input, output and formatting operations.

3 THE ASSERTIONS

Assertions are statements added to a program to describe the intent of the program, the relationships which must hold between the variables in the program, the rules by which the variables can be accessed, and other information about the program which cannot be expressed in the programming language. In short, assertions are a way in which to state a program's specifications. They are useful in program verification, in consistency checking, and for reporting unexpected behavior while the program is being tested.

When assertions are translated into executable code by a compiler or preprocessor, they are called "executable assertions." Executable assertions placed at the beginning of the program are called "initial assertions," those placed at the end of the program are called "final assertions," and those placed within the program are called "intermediate assertions." Initial assertions describe the conditions that must be satisfied when the program is entered. These conditions can be the values of certain variables, their ranges, or the relationship between the value of one vriable and the value of another (for example that X is greater than Y). Final assertions describe the result that the program is to compute--the range of values of the results and the relationships that must hold between any of the resultant variables. Intermediate assertions are used to describe the values that variables can assume and the relationship between these values and the values of other program variables at intermediate points in the program. They may also be used to specify the computational steps that a program must perform in response to the value of a particular logical expression.

Almost any condition or specification can be expressed using executable assertions. An executable assertion is a logical expression which, if evaluated to false, signals the violation of a specifiction. When the program is executed, the logical expression in an assertion is evaluated when the assertion is reached. If it is false, an error

message is printed, the assertion that was violated is recorded and a recovery routine (if specified in the assertion) is executed.

The assertions written for the test program, ORBP, describe three kinds of specifications: (1) the ranges of variables, (2) the relationships among variables, and (3) the history of the computation. For example, the assertions shown in Fig. 3.1 define the range of the parameter VALUE when it is interpreted as the angle between a point and the perigee. An example of the second type of assertion is shown in Fig. 3.2. Here VALUE is interpreted as the radius of the orbit. Therefore, its value must have a particular relation to the value of the semi-major axis, A , and the eccentricty of the orbit, E. The final type of assertion is used to keep track of the iterative computation of the angle from perigee when VALUE is interpreted as the time at which a point on the orbit is reached. The computation proceeds in two different ways depending on whether the number of iterations is even or odd. The code segment which performs this computation is shown in Fig. 3.3. The computation is limited in the number of iterations it is to perform. This is verified by adding the variable MTRY to the code to count the number of iterations and an assertion to test its value. This also helps identify errors which cause the computation to be performed out of sequence.

The assertions for the test program were organized in the following way. Initial assertions were gathered together in a logical function INPCHK which was invoked by the initial assertion

```
INITIAL ( INPCHK(MODE, VALU, ORBEL, STATE) )
```

which is the first assertion in the test program. This assertion shows that assertions can contain calls to logical functions, that is functions whose value evaluates to true or false. INPCHK contains assertions which check the ranges of the input variables to ORBP, verify the

```
ASSERT (VALUE .GE. 0.0)

ASSERT (VALUE .LE. TWOPI)
```

Figure 3.1. An Example of Range Assertions

```
ASSERT (VALUE .GE. (A * (1.0 - E) ) )

ASSERT (VALUE .LE. (A * (1.0 + E) ) )
```

Figure 3.2. An Example of Relationship Assertions

```
      T = VALUE
      EA1 = FM
      NTRY = -1
41    CONTINUE
      MTRY = NTRY
      MTRY = NTRY + 1
      IF  (NTRY .EQ. 20) GO TO 250
      EA = FM + E * SIN (EA1)
      IF  (ABS  (EA1-EA)  .LE. EMISS) GO TO 42
      IF  (MOD (NTRY,2) .EQ. 1) 45, 46
   45 CONTINUE
      EA1 = EA2 - (EA1-EA2)**2/(EA+EA2-2.*EA1)
      ASSERT  ( MTRY .LT. NTRY )
      GO TO 41
   46 EA2 = EA1
      EA1 = EA
      ASSERT (MTRY .LT. NTRY )
      GO TO 41
```

Figure 3.3. An Example of History Assertions

relationships that must hold among these variables and verifies that the orbit defined by the orbital element vector is an ellipse.

The output assertions for ORBP were written in the same way. A logical function OUTCHK was written which was invoked by the assertion

FINAL (OUTCHK(MODE, VALU, ORBEL, STATE))

just before ORBP was exited. The function OUTCHK checked the output of the test program by comparing the representation of the orbit in terms of the state vector which was calculated, to the representation of the orbit as input to ORBP in the orbital element vector. It does this by recalculating the orbital element vector from the state representation of the point on the orbit. The code and assertions for OUTCHK are shown in Appendix A.

Other assertions were added directly to the test program to check the ranges of variables, the relationships between their values and the order of the computation. These assertions were derived from documentation provided with the program and from equations from the theory of orbital mechanics. The listings of these three programs are included in Appendix A.

The assertions for ORBP were not all written at one time. In fact, the combination of existing assertions and the search algorithm made the creation of new assertions an iterative process. As more was learned about the behavior of the program through the testing process, better, more precise assertions could be written about it.

Assertions were first written from information gained by reading the program and its documentation and by studying the equations of orbital mechanics. However, the first set of grid tests identified a number of errors which could not be detected using assertions and a

number of errors which were not detected by the assertions already in the code. Therefore, the results from these tests were used to write more precise assertions which could detect these errors. No new assertions were added to the code after the first set of grid tests although a number of assertions were changed. This is discussed more in the results section below.

4 THE ERRORS

Errors were generated for the test program using a procedure developed by Brooks. A complete description of the method can be found in Gannon, Brooks and Meeson.[1] The method uses error types and frequencies from a previous study[2] to randomly select a set of errors to be "seeded" in the program. The error types from Project 5 of this study were used in the experiment. These error types or categories are shown in Table 4.1.

Not all of the categories were chosen for use in the experiment. Operation errors, other errors, documentation errors, and problem report rejection errors were not included because they did not include errors which were detectable while running the program. The experiment was specifically concerned with detecting run-time errors. Data input errors and data output errors were not included because the test program does not include any input or output statements of any consequence other than error messages. Data definition errors (which have to do with subscript referencing) were not included since explicit, constant subscripts were used to access arrays in the test program. Finally, data base errors were not included since the test program does not access a defined data base.

The remaining categories (computational errors, logic errors, data handling errors, and interface errors) were used to generate errors for ORBP. Table 4.2 shows (1) the percent of errors found in each category by the original study, (2) the percent of errors in each category when only these categories are considered, (3) the number of errors and the percent of errors in each category which were used in the study, and (4)

[1] C. Gannon, R. N. Meeson, and N. B. Brooks, An Experimental Evaluation of Software Testing, General Research Corporation CR-1-854, May 1979.

[2] Thayer et al., op. cit.

TABLE 4.1

ERROR TYPES USED IN EXPERIMENT

PROJECT 5 ERROR CATEGORIES		Applicable to Experiment
A_000	COMPUTATIONAL ERRORS	✓
A_100	Incorrect operand in equation	✓
A_200	Incorrect use of parenthesis	✓
A_300	Sign convention error	✓
A_400	Units or data conversion error	✓
A_500	Computation produces over/under flow	✓
A_600	Incorrect/inaccurate equation used/wrong sequence	✓
A_700	Precision loss due to mixed mode	✓
A_800	Missing computation	✓
A_900	Rounding or truncation error	✓
B_000	LOGIC ERRORS	✓
B_100	Incorrect operand in logical expression	✓
B_200	Logic activities out of sequence	✓
B_300	Wrong variable being checked	✓
B_400	Missing logic or condition tests	✓
B_500	Too many/few statements in loop	
B_600	Loop iterated incorrect number of times (including endless loop)	
B_700	Duplicate logic	✓
C_000	DATA INPUT ERRORS	
C_100	Invalid input read from correct data file	
C_200	Input read from incorrect data file	
C_300	Incorrect input format	
C_400	Incorrect format statement referenced	
C-500	End of file encountered prematurely	
C_600	End of file missing	
D_000	DATA HANDLING ERRORS	✓
D_050	Data file not rewound before reading	
D_100	Data initialization not done	✓
D_200	Data initialization done improperly	✓
D 300	Variable used as a flag or index not set properly	✓
D_400	Variable referred to by the wrong name	✓
D_500	Bit manipulation done incorrectly	
D_600	Incorrect variable type	✓
D_700	Data packing/unpacking error	
D_800	Sort error	
D_900	Subscripting error	

Table 4.1 (cont.)

	PROJECT 5 ERROR CATEGORIES	Applicable to Experiment
E_000	DATA OUTPUT ERRORS	
E_100	Data written on wrong file	
E_200	Data written according to the wrong format statement	
E_300	Data written in wrong format	
E_400	Data written with wrong carriage control	
E_500	Incomplete or missing output	
E_600	Output field size too small	
E_700	Line count or page eject problem	
E_800	Output garbled or misleading	
F_000	INTERFACE ERRORS	✓
F_100	Wrong subroutine called	✓
F_200	Call to subroutine not made or made in wrong place	✓
F_300	Subroutine arguments not consistent in type, units, order, etc.	✓
F_400	Subroutine called is nonexistent	
F_500	Software/data base interface error	
F_600	Software user interface error	
F_700	Software/software interface error	✓
G_000	DATA DEFINITION ERRORS	
G_100	Data not properly defined/dimensioned	
G_200	Data referenced out of bounds	
G_300	Data being referenced at incorrect location	
G_400	Data pointers not incremented properly	
H_000	DATA BASE ERRORS	
H_100	Data not initialized in data base	
H_200	Data initialized to incorrect value	
H_300	Data units are incorrect	
I_000	OPERATION ERRORS	
I_100	Operating system error (vendor supplied)	
I_200	Hardware error	
I_300	Operator error	
I_400	Test execution error	
I_500	User misunderstanding/error	
I_600	Configuration control error	

Table 4.1 (cont.)

PROJECT 5 ERROR CATEGORIES		Applicable to Experiment
J_000	OTHER	
J_100	Time limit exceeded	
J_200	Core storage limit exceeded	
J_300	Output line limit exceeded	
J_400	Compilation error	
J_500	Code or design inefficient/not necessary	
J_600	User/programmer requested enhancement	
J_700	Design nonresponsive to requirements	
J_800	Code delivery or redelivery	
J_900	Software not compatible with project standards	
K_000	DOCUMENTATION ERRORS	
K_100	User manual	
K_200	Interface specification	
K_300	Design specification	
K_400	Requirements specification	
K_500	Test documentation	
X0000	PROBLEM REPORT REJECTION	
X0001	No problem	
X0002	Void/withdrawn	
X0003	Out of scope - not part of approved design	
X0004	Duplicates another problem report	
X0005	Deferred	

TABLE 4.2

TYPES OF ERRORS USED IN THE EXPERIMENT

Error Category	Study %	Relative %	Number Selected	%	Number Used	%
A	12.1	22.2	5	20.8	4	23.5
B	24.5	44.9	5	20.8	3	17.6
D	11.0	20.1	9	37.5	7	41.2
F	7.0	12.8	5	20.8	3	17.6
	54.6	100.0	24	99.9	17	99.9

the number of errors and percent of errors in each category which were successfully detected by assertions (see results section).

In the original study, no attempt was made to match the error type or category to a specific statement type in the program. In generating errors for the experiment, statement types and other descriptive information about the test program were generated automatically using an automated program verification system, SQLAB.[1] The statement types were then matched against errors using the method outlined below.

4.1 THE ERROR SEEDING METHOD

The errors were generated in the following way. First, each statement in the test program was classified by type. Then a table matching the error categories to statement types was constructed. This is shown in Table 4.3. The set of statement types found in the test program was then added to the error-category/statement-type table. This gave a list of available error sites in the test program with associated error categories. From this list of available error sites, potential error sites were randomly selected and matched with the error subcategories by a previously written computer program.

From the list of potential sites and associated error subcategories, errors were developed. The error site was first checked to be sure that the error sub-category was appropriate for the site. For example, if error type A200 (incorrect use of parenthesis) is selected as a subcategory, the statement must contain parentheses in order to include this error.

As each error was constructed, it was included in an "error packet" containing an error number, a comment which identified the error subcategory, and the code which altered the original code of the test program in order to produce the error. Since the test program was

[1] S. H. Saib, "Application of the Software Quality Laboratory," Vol. 2 of Infotech State of the Art Report, Software Testing, Infotech International, Ltd., Maidenhead, Berkshire, England, 1979, pp. 231-243.

TABLE 4.3

RELATIONSHIPS BETWEEN SOFTWARE PROPERTIES AND ERROR TYPES

Major Error Categories and Error Types

Test Software Property	A Computational									B Logic							C Input						E Output							D Data Handling							F Interface				G Data Definition				H Data Base		
Statement	A100	A200	A300	A400	A500	A600	A700	A800	A900	B100	B200	B300	B400	B500	B600	B700	C100	C200	C300	C400	C500	C600	E100	E200	E300	E400	E500	E600	E700	D050	D100	D200	D300	D400	D600	D900	F100	F200	F300	F700	G100	G200	G300	G400	H100	H200	H300
Assignment	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓			✓			✓	✓	✓			✓	✓	✓				✓				✓	✓	✓	✓	✓	✓	✓			✓	✓	✓			
ASSIGN											✓																							✓	✓												
BACKSPACE																																		✓	✓												
CALL	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓								✓					✓	✓									✓	✓	✓	✓	✓	✓	✓		✓	✓	✓				
COMMON																																		✓	✓	✓					✓	✓	✓				
Computed GOTO											✓	✓	✓																					✓	✓												
CONTINUE											✓																																				
DATA															✓																✓	✓	✓	✓	✓	✓					✓	✓		✓	✓	✓	✓
DECODE					✓		✓		✓			✓			✓										✓								✓	✓	✓	✓						✓	✓	✓	✓	✓	✓
DIMENSION																																		✓	✓	✓					✓		✓				
DO											✓	✓	✓	✓	✓	✓																	✓	✓	✓	✓								✓			
ENCODE					✓		✓		✓			✓			✓										✓								✓	✓	✓	✓						✓	✓	✓	✓	✓	✓
ENTRY											✓																										✓		✓								
EQUIVALENCE																																		✓	✓	✓							✓				
EXTERNAL																																		✓	✓		✓		✓	✓	✓						
FORMAT							✓		✓										✓						✓	✓	✓	✓	✓																		
FUNCTION																																		✓	✓		✓			✓							
Assigned GOTO											✓	✓	✓			✓																		✓	✓												
GOTO											✓		✓																																		
IF end-of-file											✓	✓	✓			✓																	✓	✓	✓	✓	✓		✓								
Three-branch IF	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓																	✓	✓	✓	✓	✓	✓	✓			✓	✓				
Two-branch IF	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓																	✓	✓	✓	✓	✓	✓	✓			✓	✓				
Logical IF	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓																	✓	✓	✓	✓	✓	✓	✓			✓	✓				
INTEGER																																		✓	✓						✓						
LOGICAL																																		✓	✓						✓						
PRINT					✓		✓		✓			✓			✓								✓						✓				✓	✓	✓	✓						✓	✓	✓			
PROGRAM																			✓				✓											✓	✓		✓										
READ					✓		✓		✓			✓			✓		✓	✓		✓	✓	✓								✓			✓	✓	✓	✓				✓		✓	✓	✓			
REAL																																		✓	✓						✓						✓
RETURN											✓																													✓							
REWIND																														✓				✓	✓												
STOP											✓		✓																											✓							
SUBROUTINE																																		✓	✓		✓			✓							
WRITE					✓		✓		✓			✓			✓								✓	✓	✓	✓	✓	✓	✓				✓	✓	✓	✓				✓		✓	✓	✓			✓
Interface																																															
PARAMETER	✓									✓		✓											✓	✓										✓	✓	✓	✓		✓	✓			✓				✓
INVOCATION	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓													✓	✓									✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓				✓

stored on a program library maintained by CDC UPDATE[1] (a batch source text editor), the error packets could easily be inserted into the test program. Figure 4.1 shows an example of an error packet.

Next the error packets were inserted into the test program and the program was compiled and run. This was done to insure that the errors were not detected by the FORTRAN compiler, the loader or the run-time error routines of the operating system. In this way, twenty-four errors were developed for use during the testing. Table 4.4 shows each of these errors by number, the error subcategory to which it belongs and a short description of the subcategory.

Seven of the errors generated were eliminated from the testing during the grid tests since they could not be detected using assertions. This is discussed in Sec. 6.1.

```
*IDENT    13
*DELETE   ORBP.63
 C  A100
    VALUE = VALU-RBODY
```

Figure 4.1. An Error Packet

[1] UPDATE Reference Manual, Control Data Corporation, Arden Hills, Minn., 1975.

TABLE 4.4

ERRORS USED IN THE EXPERIMENT

Error Number	Category	Description
1	A200	incorrect use of parenthesis
3	A300	sign convention error
8	A600	incorrect/inaccurate equation used/wrong sequence
13	A100	incorrect operand in equation
14	A800	missing computation
28	B400	missing logic or condition tests
31	B400	missing logic or condition tests
36	B200	logic activities out of sequence
37	B200	logic activities out of sequence
40	B300	wrong variable being checked
41	D200	data initialization done improperly
46	D100	data initialization not done
47	D100	data initialization not done
48	D400	variable referred to by the wrong name
52	D600	incorrect variable type
54	D600	incorrect variable type
55	D600	incorrect variable type
56	D400	variable referred to by the wrong name
57	D300	variable used as a flag or index not set properly
62	F100	wrong subroutine called
64	F100	wrong subroutine called
67	F700	software/software interface error
74	F200	call to subroutine not made or made in wrong place
77	F700	software/software interface error

5 THE EXPERIMENT

The errors were inserted into the test program one at a time. First, grid tests were performed to identify any errors which could not be detected by assertions or errors for which other assertions had to be written. After the former errors were eliminated from consideration and assertions were added to the code to detect the latter, the grid tests were performed again. The results from these tests were used as a baseline by which to evaluate the search technique. A set of assertions which were violated when the test program was run using the grid test method was associated with each error. After the grid tests were run, the search algorithm was used to test the program by varying only three of the maximum of eight variable parameters. Finally, the search algorithm was allowed to vary all of the parameters.

Recall that of the eight parameters in the orbital element vector, only six of these are independent. The independent variables are: (1) longitude of the ascending node, (2) inclination of the orbit plane, (3) argument (angle) of the perigee, (4) eccentricity, (5) time at perigee, and (6) semi-major axis. These parameters along with MODE and VALUE were the parameters which could be varied by the test driver. For each of the tests, a standard orbit was used as a basic test case. The parameters of the orbit are shown in Table 5.1. The parameters which were not being varied in a test remained fixed at these values.

5.1 GRID TESTS

For the grid tests, three variables were varied, MODE, VALUE, and the eccentricity of the orbit. The tests were performed in the following way. The standard orbit was input to the test driver program. The test driver then varied the values of the parameters and ran tests of ORBP. The data collection routines recorded the number of assertions violated in each test along with the values of the input variables.

TABLE 5.1

STANDARD ORBITAL PARAMETERS

Parameter	Value
Longitude of the ascending node	π
Inclination of the orbit plane	$\pi/2$
Angle of the perigee	$\pi/2$
Eccentricity	0.1
Time at perigee	0
Semi-major axis	10000

The parameter values were varied in the following way. The value of the eccentricity of the orbit was varied from 0.1 to 0.9 in steps of 0.2. (The range and step size of any variable can be varied by the test driver program.) The value of the mode was then varied from 0 to 5. For each value of MODE, the corresponding VALUE parameter was varied over its range from minimum to maximum such that eleven VALUEs were generated for each value of MODE. The range of the VALUE parameter for each value of the MODE parameter is shown in Table 2.3. In this way, a coarse "grid" was drawn over the input space of the program for three variables. The values of the variables determine points in the grid and were used as input values to the program during this series of tests.

For error number 52, the time at perigee had to be varied instead of the eccentricity in order for the assertions to detect the error. This parameter was varied from 0 to the period in order to generate eleven test values.

5.2 SEARCH VARYING THREE PARAMETERS

In the second part of the experiment the search routine was used to detect the errors. Again the standard orbit was used as a basis for the testing. It was input to the test driver and the search routine was allowed to vary the values of MODE, VALUE and the eccentricity of the orbit (time at perigee in the case of error 52) in order to locate the error in the test program. All other input parameters to ORBP remained constant. T e testing was done by inserting the errors in the test program one at a time. For each error, the assertions violated were recorded along with the values of the parameters.

The search routine was allowed to run until it found the number of assertion violations preset by an input parameter. When this number of assertion violations was detected, it was increased by one and the search algorithm tried to locate a combination of input values which caused the new number of assertions to be violated. In this way, the search algorithm was directed to locate values of the input parameters which caused the maximum number of assertions to be violated. The search routine stopped if it had not located this number of errors in one hundred more tests.

5.3 SEARCH VARYING ALL PARAMETERS

For the final stage of the experiment, the search routine was allowed to vary all of the input parameters in order to locate assertion violations. Again, the standard orbit was used as a starting test case. In addition to this set of input data, the search routine chose random values for the parameters until eleven test cases were identified. A test case consisted of the orbital element vector and the MODE and VALUE parameters. This is one more test case than the number of variables in the input space of the test program and is the number of function values required to construct the complex. The number of assertions violated for each test case was determined by running the test program.

The search continued by varying the input parameters according to the search algorithm until a preset number of assertions was violated. As in the previous search tests, when this occurred the number was increased by one and the search continued in order to locate a new test case which violated this new number of assertions. If the new number of assertions were not violated in any test after one hundred tests, the search was stopped. Each one of the errors was tested in this way.

Figures 5.1 to 5.5 show some of the output produced by the search program when run with error number 13. Figure 5.1 shows a template for interpreting the output. Error information produced in response to the violation of an assertion appears first, as shown by error 9 in Fig. 5.2; or there may be none, as in test 6. Next, the test number and the action performed by the search routine in selecting the new point is printed. The possible search actions are shown in Table 5.2. The values of MODE, the orbital parameters and VALUE are then printed. Finally, the "performance value," the number of assertions violated in the test is printed.

Figure 5.2 shows the tests used to initialize the complex, that is those which determine the vertices of the complex by obtaining eleven values of the error function. Tests 7 and 9 have already caused assertions to be violated. Note that all the orbital elements, MODE and VALUE are being varied.

Figure 5.3 shows tests in the middle of the testing cycle. The search routine is applying appropriate transformations, rotation, reflection, centroid substitution and contraction in order to remove the worst point from the complex and locate a point where the maximum number of assertions are violated. Note that not all search actions are tried (e.g., expansion, shrinkage), since other parameters of the complex and error function determine which transformations are applied. In Fig. 5.3 tests 44, 45 and 47 located new input values which caused assertions to be violated, whereas test 46 did not.

Error information from assertions

Test Number		Search Action		
Worst Point		Orbital Elements		
Mode	Longitude of Ascending Node (Radians)	Inclination of the Orbit Plane (Radians)	Angle of Perigee (Radians)	Semi-latus Rectum (Meters)
Eccentricity	Time at Perigee (Seconds)	Period/2π (Seconds/ Radian)	Semi-major Axis (Meters)	Value

Performance Value = Number of assertion violations

Figure 5.1. Search Program Output Template

TABLE 5.2

POSSIBLE SEARCH ACTIONS

Search Action	Meaning
INITIAL	Initialize Complex
REFLECT	Reflection
EXPAND	Expansion
CENTROID	Centroid Substitution
CONTRACT	Contraction
SHRINK	Shrinkage
ROTATE	Rotation
RE-INITIAL	Re-initialize Complex

```
TEST=    5      INITIAL
WORST POINT
      1.000000000          2.228417351          .4126283562E-01      5.619764999          7380198.092
      .4903670443          4610.762172          1517.131149          9716666.034          9746388.950
PERFORMANCE VALUE=0.

TEST=    6      INITIAL
WORST POINT
      2.000000000          5.179884825          .1473052791          3.408350459          10446559.90
      .5840825780          18085.35416          3162.494392          15855807.66          15708809.93
PERFORMANCE VALUE=0.

ORBP HAS REPLACED IMPOSSIBLE RADIUS WITH PERIGEE -

R-RBODY=-4086816.74491795897         VALUE= 2477545.65879538655
R= 2288363.25508202612          RBODY= 6375180.00000000000

TEST=    7      INITIAL
WORST POINT
      4.000000000          4.219872378          .6978480303          5.440010710          4010669.367
      .7526366751          188.7583587          1409.391651          9251020.763          2477545.659
PERFORMANCE VALUE= 2.000000000

TEST=    8      INITIAL
WORST POINT
     0.                    .9704060213E-01      .8554524602          1.173598384          8912339.696
      .5692411226          2716.512589          2397.998640          13184625.99          3.630148349
PERFORMANCE VALUE=0.

ORBP HAS REPLACED IMPOSSIBLE RADIUS WITH PERIGEE -

R-RBODY=-1158666.49469146132         VALUE= 9849931.05953472853
R= 5216513.50530853868          RBODY= 6375180.00000000000

TEST=    9      INITIAL
WORST POINT
      5.000000000          3.285706144          1.956802017          5.848423395          8371219.419
      .6047537135          2341.114231          2401.684845          13198134.13          9849931.060
PERFORMANCE VALUE= 2.000000000

TEST=   10      INITIAL
WORST POINT
      1.000000000          5.689950305          .1382653909          3.279775555          26473235.12
      .2577490650          35616.80710          7563.809829          28357129.09          22207982.01
PERFORMANCE VALUE=0.

TEST=   11      INITIAL
WORST POINT
      2.000000000          .7720035681          .6434594192          5.510889374          32954922.79
      .2318260222          62045.79446          10294.71221          34826620.47          32069696.10
PERFORMANCE VALUE=0.
```

```
ORBP HAS REPLACEL IMPOSSIBLE RACIUS WITH PERIGEE -

R-RBODY=-2739711.36942151189          VALUE= 4044234.17143005133
R= 3635468.63057847321          RBODY= 6375180.00000000000

TEST=   44     ROTATE
WORST POINT
      4.000000000         3.946273241         .8606307614         4.763729021         6170490.267
      .6973025746         5202.081546         2084.849562         12010239.68         4044234.171
PERFORMANCE VALUE= 2.000000000

ORBP HAS REPLACED IMPOSSIBLE RACIUS WITH PERIGEE -

R-RBODY=-3559778.07904553413          VALUE=0.
R= 2815401.92095445096          RBODY= 6375180.00000000000

TEST=   45     REFLECT
WORST POINT
      5.000000000         .9642159314         3.044470996         6.283185308         4570691.766
      .6234597739         6434.586072         1024.096180         7477028.285        0.
PERFORMANCE VALUE= 2.000000000

TEST=   46     CENTROID
WORST POINT
      3.000000000         3.622123975         1.328380598         3.749728861         12748680.77
      .5468299210         18160.82256         3884.985419         18187016.25         24410.08331
PERFORMANCE VALUE=0.

ORBP HAS REPLACED IMPOSSIBLE RADIUS WITH PERIGEE -

R-RBODY=-1051749.44438433647          VALUE= 4533190.34480997920
R= 5323430.55561566353          RBODY= 6375180.00000000000

TEST=   47     CONTRACT
WORST POINT
      4.000000000         2.293169953         2.186425797         5.016457085         8438408.516
      .5851448474         12297.70432         2302.448197         12832022.27         4533190.345
PERFORMANCE VALUE= 2.000000000
```

Figure 5.3. Intermediate Stage of Search Testing

```
ORBP HAS REPLACED IMPOSSIBLE RADIUS WITH PERIGEE -

R-RBODY= 2437519.01370167732          VALUE= 11674092.7922087908
R= 8812699.01370167732          RBODY= 6375180.0000000000

TEST=   94     ROTATE
WORST POINT
       4.000000000          2.272290885          1.428911545          3.575033478          13179367.00
       .4954972341          19425.43816          3656.918571          17468088.60          11674092.79
PERFORMANCE VALUE= 2.000000000

R-RBODY= 6110497.79315781593          VALUE= 18860857.7931579351
R= 12485677.7931578159          RBODY= 6375180.0000000000

TEST=   95     REFLECT
WORST POINT
       5.000000000          .5598999719          2.837951677          6.283185308          3954322.723
       .8433065145          15942.72284          2537.363146          13690635.60          18860857.79
PERFORMANCE VALUE= 1.000000000

ORBP HAS REPLACED IMPOSSIBLE RADIUS WITH PERIGEE -

R-RBODY= 2027068.48747164011          VALUE= 11115483.7344506383
R= 8402248.48747164011          RBODY= 6375180.0000000000

TEST=   96     CENTROID
WORST POINT
       4.000000000          2.426831755          1.579551477          4.021620299          13147715.03
       .5647853136          23299.68466          4248.984817          19305985.64          11115483.73
PERFORMANCE VALUE= 2.000000000

R-RBODY= 2237810.76380419731          VALUE= 14988170.7638042569
R= 6612990.76380419731          RBODY= 6375180.0000000000

TEST=   97     CONTRACT
WORST POINT
       4.000000000          1.493365863          2.208751577          5.152402804          8320417.305
       .7040459140          19621.20375          3356.652825          16498310.62          14988170.76
PERFORMANCE VALUE= 1.000000000
```

Figure 5.4. Later Stage of Search Testing

********** FINAL REPORT **********

#RUN	INPUT1	INPUT2	#FALSE ASSERTION	#DIFFERENT ASSERTION	MODE	VALUE
7	.7526	0.	2	2	4	2477545.659
9	.6048	0.	2	2	5	9849931.060
12	.2700	0.	2	2	4	13958923.49
13	.9000	0.	1	1	5	24389119.03
24	.2899	0.	2	2	4	8871067.739
25	.3879	0.	2	2	5	1760571.330
30	.2910	0.	2	2	5	20758872.74
34	.7346	0.	1	1	4	22330022.80
35	.1852	0.	2	2	5	27015515.91
37	.3555	0.	2	2	4	19513234.41
44	.6973	0.	2	2	4	4044234.171
45	.6235	0.	2	2	5	0.
47	.5851	0.	2	2	4	4533190.345
49	.9000	0.	2	2	5	0.
51	.7234	0.	2	2	4	4533190.345
53	.9000	0.	2	2	5	5737662.000
55	.7234	0.	2	2	4	7402021.345
63	.7053	0.	2	2	5	0.
65	.6261	0.	2	2	4	4533190.345
73	.7474	0.	2	2	5	0.
75	.6471	0.	2	2	4	4533190.345
83	.8071	0.	2	2	5	0.
85	.6769	0.	2	2	4	4533190.345
86	.1774	0.	1	1	4	23124989.06
87	.9000	0.	2	2	5	1415222.244
89	.7228	0.	2	2	4	5588024.749
90	.9000	0.	2	2	5	1939816.571
91	.1557	0.	2	2	4	6107643.223
92	.5503	0.	2	2	4	9951144.293
93	.3530	0.	2	2	4	8029393.758
94	.4955	0.	2	2	4	11674092.79
95	.8433	0.	1	1	5	18860857.79
96	.5648	0.	2	2	4	11115483.73
97	.7040	0.	1	1	4	14988170.76
98	.6108	0.	1	1	4	17226371.99
99	.2554	0.	2	2	5	3876077.474
100	.5485	0.	2	2	4	11161715.66
101	.4019	0.	2	2	5	7518896.567
102	.1000	0.	2	2	5	7331062.939

INPUT1 = ORBIT(6) INPUT2 = INPUT3 =

MODULE	STMT#	TYPE	FAILURES*
ORBP	109	ASSERT	34
OUTCHK	142	ASSERT	38

* HOW MANY RUNS EACH ASSERTION FAILED IN 102 RUNS

Figure 5.5. Summary of Search Testing for Error 13

Figure 5.4 shows a later stage in the search testing. Here almost every test results in some assertions being violated.

Figure 5.5 is a summary of the results of the search testing for error number 13. Only the tests in which assertions were violated are shown. The summary shows for each test (1) the number of assertions violated, (2) the number of different assertions violated, and (3) the values of MODE and VALUE. (The INPUT1 and INPUT2 columns are used for the grid testing.) The figure also shows the progress of the search routine during the testing. At the beginning of the testing, assertions were violated only every few tests. At the end of the testing, almost every test resulted in assertions being violated. Finally, the location of the assertions that were violated and the number of times that they were violated are printed.

6 RESULTS OF THE EXPERIMENT

Four major results were found from the experiments: (1) the original set of grid tests found that a number of the errors could not be detected through the use of assertions, (2) the search tests located assertion violations for two errors which the grid tests did not discover but there were two errors for which the grid tests found assertion violations where the search tests did not, (3) the search tests were more efficient than the grid tests in locating assertion violations, and (4) the search tests discovered a number of boundary conditions which caused assertion violations.

6.1 ERROR DETECTION USING ASSERTIONS

Of the twenty-four errors originally used for the testing, only nine (37.5%) of these errors were detected by the first assertions placed in the code. By adding more assertions to the test program, eight more errors were detected (33.3%). The remaining seven errors (29.2%) could not be detected by placing assertions in ORBP. Table 6.1 lists these errors along with their categories, short descriptions and the reason they could not be detected by assertions.

Two of the errors could not be detected by the test method because they occurred only after another error had occurred first. Another error occurred only if values of the input parameters were out of range, a possible source of error, but not one considered in the experiment. Three of the errors could be detected by static analysis techniques such as variable initialization checks, parameter checks and cross-references but are less easily detected using assertions. These errors cannot be easily caught by assertions because of the limits placed on the assertions by the semantics of the programming language. For example, there is no way to state in an assertion that a variable has been initialized to a particular value other than by stating that the variable has that value. If the value happens to be zero, and the compiler assigns this value to the variable automatically, then there is

TABLE 6.1

ERRORS NOT DETECTED BY ASSERTIONS

Error	Category	Description	Reason For Not Being Detected
8	A600	Variables assigned values in incorrect order	An error must occur for this section of code to be executed
36	B200	Test and branch statement deleted	Checks for out of range input values
40	B300	Variable name mispelled in computed goto	Difficult to state an assertion for this error
46	D100	Data statement deleted	Fortran compiler initializes all variables to zero
55	D600	Real variable declared as integer	Difficult to state an assertion for this error
62	F100	Subroutine call out of place	An error must occur for the section of code to be executed
77	F700	Wrong number of arguments in subroutine call	Difficult to state an assertion for this error

no way to write an assertion that states that the variable was initialized. Stated another way, we can write an assertion which states that a variable is equal to a certain value, but not that it has been initialized. Similarly, it is difficult for an assertion to state that a subroutine call has a certain number of parameters, or that a variable is spelled correctly. Since assertions are written using the constructs of the programming language, they cannot state things about the program that cannot be stated in the programming language.

The other error which was not used caused a run-time error to occur in a library routine. This could be detected in the library routine by an assertion, but not in the test routine. Again, the specific error indicates the limited power of assertions. In this case, a REAL variable was declared as INTEGER. There is no way using assertions to state that the type of a variable is REAL. Again, this error might have been located by a static analysis check for invalid parameter types.

6.2 EFFECTIVENESS OF THE SEARCH TECHNIQUES

For most errors, the search technique (using three parameters and all parameters) identified the same assertion violations as the grid testing technique. In four cases (errors 14, 28, 47, and 74), however, this did not occur. For two of the errors (28 and 47), the search technique did not identify as many assertion violations as the grid technique. In the other two cases (errors 14 and 74), the search technique identified assertion violations that were not detected in the grid tests.

In error 28, a statement is deleted which tests for a zero divisor. The sequence of code and the assertion that is violated is shown in Fig. 6.1. The statement

```
IF(X2 .LE. 0) GOTO 48
```

```
42  X2 = 1. + COS (EA)
    Q  = PI
    IF (X2 .LE.O.) GO TO 48
    ASSERT ( X2 .GT. 0.0 )
    X1 = SQRT ( (1. + E) / (1. - E) ) * SIN (EA)
    Q = 2. * ATAN2 (X1, X2)
48  CONTINUE
```

Figure 6.1. Error 28

---

which was deleted to cause the error, is used to prevent a zero divisor in the call to the arctangent subroutine. The documentation with this system support routine states that the sum of the parameters (X_1 and X_2) squared must not be equal to zero, and that the arctangent of X_1 divided by X_2 is computed (see Fig. 6.2). An assertion violation is detected by the grid test for this error but by neither of the search tests (three-parameter or all-parameter). The reason for this is that the grid test uses values for the time paramenter which locate the point on the orbit as being at apogee whereas neither of the search tests used this value. For the apogee point, the value of the angle EA becomes equal to PI and the value of X2 becomes 0 (see Fig. 6.3). No run-time error was detected by the arctangent routine for this value.

Error 47 is the deletion of a data statement. This statement initializes the value of the error tolerance for the iterative computation of the angle from perigee when the VALUE parameter indicates time. The statement which this effects and the assertions violated are shown in Fig. 6.4. Since the FORTRAN compiler initializes all variables to

$$Y = \mathrm{ATAN2}\ (X_1\ ,\ X_2)$$

Function: Computes arctangent of X_1/X_2

Constraint: $X_1^{\ 2} + X_2^{\ 2} \neq 0$

Figure 6.2. Arctangent Function

Statement

```
     X2 = 1. + COS (EA)
for  EA = π :
     X2 = 1. + COS (π)
     X2 = 1. + (-1)
     X2 = 0
```

Figure 6.3. Value of Divisor at π

Data statement deleted

```
DATA EMISS / 1.E-7 /
```

Loop exit statement

```
IF ( ABS (EA1-EA) .LE. EMISS) GO TO 42
```

Assertions violated

```
ASSERT ( ABS (EA-EA1) / (EA1-EA2) .LT. 1.0)
ASSERT ( ABS (EA+EA2 - 2.0 *EA1)  .GT. 0.0)
```

Figure 6.4. Error 47

zero, this variable is by default initialized to zero also. This changes the termination condition of the loop so that it only ends if the value of EA equals the value of EA1. Again, both assertions will be violated only if the computation is being performed for a particular point on the orbit, apogee. In this case, both the grid test and the search using three-parameters found values of the input parameters which violated both assertions. The all-parameter search did not locate a value which violated the second assertion.

Error 14 is caused by the deletion of a statement. In this case however, the three-parameter search found one more assertion violation (2) than the grid test technique (1), and the all-parameter search found one more assertion violation than the three-parameter search (3). Figure 6.5 shows the sequence of statements and the assertions associated with this error. By removing the statement

```
Q = ACOS (QPRIME)
```

error 14 causes the value of Q to be undefined. This error is detected by the assertions in the OUTCHK routine when the orbits described by the

---

Code Segment

```
QPRIME = ADIV(P-R, R*E)
Q = ACOS (QPRIME)
```

Assertions violated

```
ASSERT ( RELERR(A, ORBIT(9) ) .GE. - EPS )
ASSERT (RELERR(A, ORBIT(9) ) .LE. EPS)
ASSERT (OE(4) .LE. TWOPI + TWOPI)
```

Figure 6.5. Error 14

initial orbital parameters and the state vector representation are compared. The grid test technique located values in the input space which caused the first assertion to be violated. That is, the semi-major axes of the two orbits did not agree. The three parameter search located other values for which this was also true and caused the second assertion to be violated. The all-parameter search, since it also varied the value of the argument of the perigee in the orbit element vector, was able to locate values in the input space which caused the third assertion to be violated.

Error 74 is caused by the deletion of a call to a subroutine which copies the input orbital element vector to another array. Assertions were written to compare the values of these two arrays after the point of the call in the code. The grid test technique and the three-parameter search detected assertion violations for all of the variables in the orbital element vector except one. This value was equal to 0 in the original orbital element description and was not varied by the two test methods. Since the FORTRAN compiler initialized the values of the receiving array to zero, the fact that this variable was not copied was not detected. When the all-parameter search was allowed to vary this parameter, the assertion violation for this parameter occurred also. Figure 6.6 shows the code and assertions for this error.

Table 6.2 summarizes the results for these errors, showing the error number and the number of assertion violations detected by each of the three testing methods.

6.3 EFFICIENCY OF THE SEARCH METHOD

Data which could be used to measure the efficiency of the search methods relative to the grid testing method were not collected during the experiment. However, a rough estimate of the relative efficiencies of the two methods is shown in Table 6.3. Except in the case of error 52, in which it took 683 tests to perform the entire grid test, all of the errors required 317 tests. Table 6.3 shows for each error, the

Statement

```
CALL XMIT (8,ORBEL(2),OE(2))
```

Assertions violated

```
ASSERT  ( OE(2) .EQ. ORBEL(2) )
ASSERT  ( OE(3) .EQ. ORBEL(3) )
ASSERT  ( OE(4) .EQ. ORBEL(4) )
ASSERT  ( OE(5) .EQ. ORBEL(5) )
ASSERT  ( OE(6) .EQ. ORBEL(6) )
ASSERT  ( OE(7) .EQ. ORBEL(7) )
ASSERT  ( OE(8) .EQ. ORBEL(8) )
ASSERT  ( OE(9) .EQ. ORBEL(9) )
```

Figure 6.6. Error 74

TABLE 6.2

ASSERTION VIOLATIONS DETECTED BY EACH TESTING METHOD

Error Number	Number of Invalid Assertions Detected by Testing Technique		
	Grid	3-Variable Search	All-Variable Search
14	1	2	3
28	1	0	0
47	2	2	1
74	7	7	8

TABLE 6.3

FIRST ASSERTION VIOLATIONS DETECTED BY ALL-VARIABLE SEARCH

Error Number	Test Number of First Assertion Violation
1	5
3	2
13	7
14	5
28	*
31	4
37	5
41	3
47	57
48	3
52	3
54	3
56	5
57	7
64	2
67	5
74	2

*No assertion violations detected

number of the test in which the all-parameter search testing technique detected the first assertion violation. This table shows that for 15 of the 17 errors the all-parameter search technique detected the first assertion violation on or before the seventh test.

6.4 SPECIAL CASES

During the experiment a number of assertions were revised. These assertions were changed because of the results from the search tests. In three cases, the search technique discovered input values for which assertions were violated. In each case it was later discovered that the assertions were incorrect. These special values were not discovered by the grid testing technique. This illustrates one important result of the testing method, that the development of assertions and the testing occur as a coupled iterative process. The original assertions help to locate errors, the search technique locates new assertion violations which are either errors in the software or in the assertions. Throughout the testing process, the accuracy of the assertions was improved along with the ability to detect errors.

The first assertion which was discovered as being incorrect was one which checks the value of the angle from perigee (FM) computed from the time (VALUE), time at perigee (TP) and period (PP). The code, the original assertion and the corrected assertion are shown in Fig. 6.7. This assertion violation was found by the all-parameter search by varying the time at perigee (TP). This caused the value of the angle from perigree to become negative. The time at perigree had not been varied by either of the other two test methods.

The second incorrect assertion was found by the three-parameter search method. This assertion violation was due to the nature of the orbital descriptions and the inherent inaccuracy of the calulations. In the orbital descriptions, a value of 2π is equivalent to 0, or stated another way, an orbit which begins at perigee angle equal to 0, is again

Original Code and Assertions

```
FM = (VALUE -TP) / PP
ASSERT ( FM .GE. 0.0)
ASSERT ( FM .LE. TWOPI +EPS)
```

Revised Code and Assertion

```
FM = (VALUE -TP) / PP
ASSERT ( ABS(FM) .LE. TWOPI + EPS)
```

Figure 6.7. First Incorrect Assertion

at perigee when the angle is 2π . To further compound the problem, inaccuracies in the machine representation of values and errors accumulated over the computation give rise to situations in which the value of variables is very close to 2π but not exactly 2π . Therefore the assertions which check these conditions must take this into account.

The problem becomes evident when checking the output from the test program. It is necessary to determine if the point described by the state vector is on the part of the orbit where the radius is increasing or the part where the radius in decreasing. (See Fig. 6.8.) This result is compared with the value of the MODE parameter when the VALUE parameter is interpreted as radius (MODE equal to 1 or 2) or altitude (MODE equal to 4 or 5).

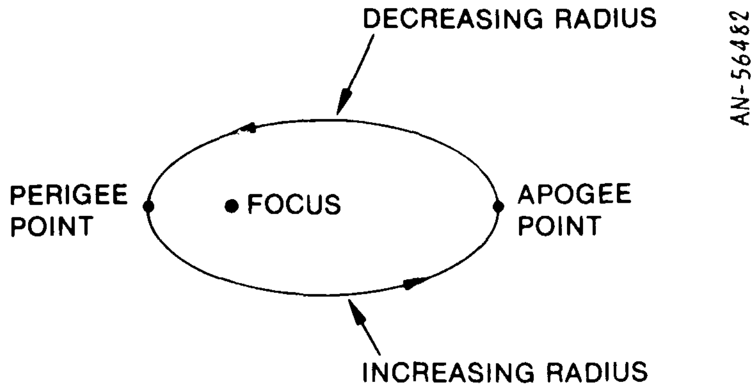

Figure 6.8. Increasing and Decreasing Radii

The point can be located on the increasing (M=1) or decreasing (M=2) radius by comparing the time at apogee (TA), the time at perigee (TP), and the time given in the state vector (TS). This is done in the code seqment in Fig. 6.9. In order to make this calculation correctly, it is not only necessary to correct for the fact that 0 equals 2π but also for the case in which the calculations give results very close to these values. These corrections are also shown in Fig. 6.10.

Another interesting result revealed by this assertion was that the calculation of the state vector time (TS) was not corrected to be less than or equal to the period. This is a quirk of an algorithm in a support routine and was not revealed by the documentation. Again, the search routine identified input values which caused these assertions to be violated. It is difficult to see how test cases could have been constructed to illustrate these errors.

```
IF ( TP .EQ. 0.0 )
   IF (TS .GE. TP .AND. TS .LE. TA)
       M = 1
   ELSE
       M = 2
   ENDIF
ORIF ( TA .EQ. 0.0 )
   IF (TS .GT. TA .AND. TS .LT. TP)
       M = 1
   ELSE
       M = 2
   ENDIF
ORIF ( TP .GT. TA )
   IF ( TS .GT. TA .AND. TS .LT. TP )
       M = 2
   ELSE
       M = 1
   ENDIF
ELSE
   IF (TS .GE. TP .AND. TS .LE. TA)
       M = 1
   ELSE
       M = 2
   ENDIF
ENDIF
```

Figure 6.9. Code to Locate Point on Radius

Corrections for time greater than period

```
TP = AMOD(ORBIT(7), PERIOD)
TA = AMOD(TP+ORBIT(8)*PI, PERIOD)
TS = AMOD (STATE(1), PERIOD )
```

Corrections for time close to period

```
IF ( PERIOD - TP .LE. DELTAT )
    TP = 0.0
ENDIF
IF ( PERIOD - TA .LE. DELTAT )
    TA = 0.0
ENDIF
IF ( PERIOD - TS .LE. DELTAT )
    TS = 0.0
ENDIF
```

Assertion to check MODE

```
IF ( TS .NE. TA) .AND. (TS .NE. TP) )
    ASSERT ( MODE .EQ. M )
END IF
```

Figure 6.10. Checking the Value of the MODE Parameter

The final assertion inconsistency also had to do with errors accumulated over computations and the fact that 0 is equal to 2π. This error arose in checking the angle from perigee, which is calculated when MODE equals 0. The angle from perigee is calculated from the input orbital elements and the radius. The radius can be calculated from the output state vector representation. This calculated value is then compared with the original value as input to ORBP in the VALUE parameter. The code to calculate the angle from perigee and the modified assertions are shown in Fig. 6.11. These assertions take into account that the calculated value may differ slightly from the original value and that 0 and 2π are equivalent. This inconsistency was discoverd by the all-parameter search technique.

Code Segment

```
Q = (ORBIT(5) / R - 1.0 ) / ORBIT(6)
Q = (ACOS ( Q )
```

Corrections for angle near 2π

```
IF ( ABS(TWOPI-Q) .LE. DTHETA )
    Q = TWOPI
ENDIF
IF ( ABS(TWOPI-VALUE) .LE. DTHETA )
    VALUE = TWOPI
ENDIF
```

Assertions Violated

```
ASSERT (AMOD(Q,TWOPI) .GE. AMOD(VALUE,TWOPI)-DTHETA)

ASSERT (AMOD(Q,TWOPI) .LE. AMOD(VALUE,TWOPI)+DTHETA)
```

Figure 6.11. Checking the Angle from Perigree

7 DISCUSSION

The results from the experiment show that it is possible to detect errors automatically using assertions and search techniques. The major limitation of the technique as we see it is the difficulty in writing the assertions. The number of assertions which need to be written, the conditions they should describe and where they should be placed are all questions which are difficult to answer. In addition, the assertions are difficult to write and the task of writing them is not pleasant. On the other hand, the search testing technique aids in the refinement of the assertions.

Unfortunately, our results have also shown the limitations of assertions. There is sometimes no way to easily express exactly what is wanted by using the current semantics. In some cases, it seems that other techniques are more suited to detecting certain types of errors.

One may also argue with the technique of "error seeding," but we believe it to be a very effective way in which to control some of the problems in an experiment such as this. Using programs from actual development efforts containing unknown errors would introduce factors into the experiment which could not be controlled. Interpreting the results of such an experiment would therefore be more difficult.

Equating assertion violations with errors is also a point which may be argued. In this experiment, it was assumed that once an assertion violation was detected, the error would become self-evident. This is obviously not the case. This will be true only if assertions are placed in the correct spot and describe the nature of the error. Again, only further experimentation can determine how useful the technique is at locating errors.

The way in which the error function was constructed to allow the search routine to be used can also be questioned. Simply summing the

number of assertions to determine the value of the function is a crude technique. The search technique is thereby driven to select input values which maximize the number of assertions violated. We have found some evidence to indicate that errors are not randomly distributed; that they occur in groups. Therefore, searching for maximums of the error function should locate most of the errors in a program. However, this is still a crude method. We are investigating a method which takes the content of the assertions into account in generating new input values. This technique is taken from artificial intelligence research and will be the basis for further experiments.

In addition to the new experiments described above, we also believe that the techniques need to be applied to cases where more than one error occurs in the software, and to types of programs other than arithmetic computations (e.g. compilers). The efficiency of the technique relative to other types of testing should also be investigated.

We believe that the experiment successfully demonstrated the value of the search testing method. We were able to locate errors in a program automatically and relieved ourselves of the necessity of inventing test cases. In addition, the technique identified errors in our conception of the operation of the program as embedded in the assertions.

---

[1] Benson, op. cit.

Bibliography

Alberts, D. S., "The Economics of Software Quality Assurance" in AFIPS Conference Proceedings: 1976 National Computer Conference, Vol. 45, AFIPS Press, Montvale, N.J., pp. 433-442.

Benson, J., A Preliminary Experiment in Automated Software Testing, General Research Corporation TM-2308, February 1980.

Box, M. J., "A New Method of Constrained Optimization and Comparison with Other Methods," Computer Journal, Vol. 8, 1965.

Clarke, L. A., "A System to Generate Test Data and Symbolically Execute Programs," IEEE Transactions on Software Engineering, Vol. SE-2, September 1976.

Cooper, D. W., Adaptive Learning Requirements and Critical Issues, General Research Corporation CR-4-708, January 1977.

Control Data Corporation, Update Reference Manual, Arden Hills, Minnesota, 1975.

Denn, M. M., Optimization by Variational Methods, McGraw-Hill, 1969.

Fries, M. J., Software Error Data Acquisition, Boeing Aerospace Company, RADC-TR-77-130, Seattle, Washington, April 1977.

Gannon, C., "Error Detection Using Path Testing and Static Analysis," Computer, Vol. 12, August 1979.

Gannon, C., R. N. Meeson, and N. B. Brooks, An Experimental Evaluation of Software Testing, General Research Corporation CR-1-854, May 1979.

Howden, W. E., "Theoretical and Empirical Studies in Program Testing," IEEE Transactions on Software Engineering, Vol. SE-4, July 1978.

Logicon, Verification and Validation for Terminal Defense Program Software: The Development of a Software Error Theory to Classify and Detect Software Errors, Logicon HR-74012, May 1974.

Macko, S. J., Stellite Tracking, John F. Rider Publisher, Inc. New York, 1962.

Plambeck, T., The Compleat Traidsman, General Research Corporation IM-711/2, revised edition, September 1969.

Saib, S. H., "Application of the Software Quality Laboratory," INFOTECH State of the Art Report, Software Testing, Vol. 2, Infotech International Limited, Maidenhead, Berkshire, England, 1979, pp. 231-243.

Thayer, T. A., et al., Software Reliability Study, TRW Defense and Space Systems Group, RADC-TR-76-238, Redondo Beach, Calif., August 1976.

APPENDIX A

Program Listings

```
SEG NEST SOURCE

        C
              LOGICAL FUNCTION   INPCHK ( MODE, VALUE, ORBIT, STATE )
        C
        CASON
        CMODN INPCHK
        CXCOM $
        C
        C          CHECK INPUT ARGUMENTS FOR SUBROUTINE @ORBP@
        C
        C     CONCON
              COMMON / CONCON /
             1   PI, SRD, SLV, SMF, SKP, RBODY,
             2   GACC, GCON, WBODY, RHOZRO, TWOPI, HAFPI
        CCOMPKG.CONCON
              REAL    ORB(10)
              INTEGER   MODE                  $   INDICATES DATATYPE OF @VALUE@
              REAL      VALUE                 $   ANGLE, RADIUS, TIME, OR ALT.
              REAL      ORBIT(10)             $   AN ELLIPTICAL ORBIT
              REAL      STATE(10)             $   OUTPUT STATE VECTOR
        C
        C     TCONST
              DATA    EPS       / 1E-6 /
              DATA    DELTAT    / 1E-2 /     $    ABSOLUTE TIME TOLERANCE
              DATA    DTHETA    / 1E-5 /
        CCOMPKG.TCONST
        C
              RELERR(X,Y) = ABS(ABS(X) - ABS(Y)) / AMAX1( ABS(X), ABS(Y) )
        C
              CALL XMIT ( 10, ORBIT, ORB )
              INPCHK = .TRUE.
        C
              INITIAL ( MODE .GE. 0  .AND.  MODE .LE. 5 )
              FAIL ( INPCHK = .FALSE. )
        C
              CASE OF ( MODE )
     1  C     *
              CASE ( 0 )                      $   VALUE IS ANGLE
     1        .   INITIAL (VALUE .GE. 0)
     1        .   INITIAL ( VALUE .LE. TWOPI )
     1        .   FAIL ( INPCHK = .FALSE. )
     1  C     *
              CASE ( 1, 2 )                   $   VALUE IS RADIUS
     1        .   A = ORBIT(9)
     1        .   E = ORBIT(6)
     1        .   INITIAL ( VALUE .GE. A * ( 1.0 - E ) )
     1        .   INITIAL ( VALUE .LE. A * ( 1.0 + E ) )
     1        .   FAIL ( INPCHK = .FALSE. )
     1  C     *
              CASE ( 3 )                      $   VALUE IS TIME
     1        .   PERIOD = ORBIT(8) * TWOPI
     1        .   INITIAL ( VALUE .GE. 0.0 )
     1        .   INITIAL ( VALUE .LE. PERIOD + DELTAT )
     1        .   FAIL ( INPCHK = .FALSE. )
     1  C     *
              CASE ( 4, 5 )                   $   VALUE IS ALTITUDE
     1        .   A = ORBIT(9)
     1        .   E = ORBIT(6)
     1        .   INITIAL ( VALUE .GE. A * ( 1.0 - E ) - RBODY )
     1        .   INITIAL ( VALUE .LE. A * ( 1.0 + E ) - RBODY )
     1        .   FAIL ( INPCHK = .FALSE. )
```

```
SEQ NEST SOURCE

     1    C      *
                 END CASE
          C
                 INVOKE ( ELLIPSE )
          C
                 BLOCK ( INPCHK = .FALSE. )
                 END BLOCK

                 BLOCK ( ELLIPSE )
     1    C      *       VERIFIES THAT ORBITAL VECTOR @ORB@ IS AN ELLIPTIC ORBIT
     1           .    ASSERT ( ORB(2) .GE. 0.0 )
     1           .    ASSERT ( ORB(2) .LE. TWOPI + EPS )
     1           .    ASSERT ( ORB(3) .GE. 0.0 )
     1           .    ASSERT ( ORB(3) .LE. PI )
     1           .    ASSERT(ORB(4) .GE. 0.0)
     1           .    ASSERT (ORB(4) .LE. TWOPI + EPS )
     1           .    ASSERT ( ORB(5) .GE. ABS ( ORB(9) * ( 1.0 - ORB(6) ** 2 ) - EPS ))
     1           .    ASSERT ( ORB(5) .LE. ABS ( ORB(9) * ( 1.0 - ORB(6) ** 2 ) + EPS ))
     1           .    ASSERT ( ( 1.0 - ORB(6)**2 ) .LT. 1.0 )
     1           .    ASSERT ( ORB(8) .GE. ORB(9) * SQRT ( ORB(9) / GCON ) - EPS )
     1           .    ASSERT ( ORB(8) .LE. ORB(9) * SQRT ( ORB(9) / GCON ) + EPS )
     1           .    ASSERT ( RELERR(ORB(9)**3, GCON*ORB(8)**2) .LE. EPS)
     1           .    FAIL ( PRINT ELEMENTS )
     1           .    ASSERT ( ORB(6) .GT. 0.0  .AND.  ORB(6) .LT. 1.0 )
                 END BLOCK

                 BLOCK ( PRINT ELEMENTS )
     1           .    WRITE (6,1000) ORB(9), GCON, ORB(8)
     1      1000 .    FORMAT ( * SEMI-MAJOR AXIS = * G24.18 /
               1.              * GCON = * G24.18 / * PERIOD / 2 PI = * G24.18 )
                 END BLOCK
          C
                 RETURN
                 END
*****************************************************************************************
```

SEQ NEST SOURCE LOGICAL FUNCTION OUTCHK (MODE, VALUE, ORBIT, STATE)

```
             LOGICAL FUNCTION   OUTCHK ( MODE, VALUE, ORBIT, STATE )
       C
       CASON
       CMODN OUTCHK
       CXCOM $
       C
       C          CHECKS FINAL CONDITIONS FOR SUBROUTINE GORBPG
       C
       C     CONCON
             COMMON / CONCON /
            1   PI, SRC, SLV, SMF, SKP, RBODY,
            2   GACC, GCON, WBODY, RHOZRO, TWOPI, HAFPI
       CCOMPKG.CONCON
             INTEGER   MODE                  $   INPUT MODE FLAG
             REAL      VALUE                 $   INPUT VALUE PARAMETER
             REAL      ORBIT(10)             $   INPUT ORBITAL ELEMENT VECTOR
             REAL      STATE(10)             $   OUTPUT STATE VECTOR
       C
       C     TCONST
             DATA    EPS       / 1E-6 /
             DATA    DELTAT    / 1E-2 /      $   ABSOLUTE TIME TOLERANCE
             DATA    DTHETA    / 1E-5 /
       CCOMPKG.TCONST
       C
       C
       C
       C
             RELERR(X,Y) = (X - Y) / Y
       C
             OUTCHK = .TRUE.
       C
       C        VERIFY THAT THE STATE VECTOR REPRESENTS A FEASIBLE
       C        POSITION ON THE ORBIT.
       C
             R = XMAG( STATE(2) )            $   R = RADIUS OF STATE VECTOR
       C
             ASSERT ( ABS(R/ORBIT(9) - 1.0) .LE. ORBIT(6) + EPS )
             ASSERT ( ABS(R/ORBIT(9) - 1.0) .GE. -ORBIT(6) - EPS )
       C        SEE PAGE 75 OF NOTES
       C
             V = XMAG( STATE(5) )            $   V = VELOCITY COMPONENT
             A = R / ( 2.0 - R * V**2 / GCON )
       C        SEE PAGE 81 OF NOTES
       C
             ASSERT ( RELERR(A, ORBIT(9)) .GE. -EPS )
             FAIL ( SEMI-MAJOR AXIS )
             ASSERT (RELERR(A, ORBIT(9)) .LE. EPS)
             FAIL ( SEMI-MAJOR AXIS )
       C
             G = XMAG( STATE(8) )            $   G = ACCELERATION COMPONENT
       C
             ASSERT ( G .GE. GCON / R ** 2 - EPS )
             ASSERT ( G .LE. GCON / R ** 2 + EPS )
       C
             PERIOD = TWOPI * ORBIT(8)
             TP = AMOD(ORBIT(7), PERIOD)
             TA = AMOD(TP+ORBIT(8)*PI, PERIOD)      $ TIME AT APOGEE
             TS = AMOD (STATE(1), PERIOD )
             IF ( PERIOD - TP .LE. DELTAT )
    1        .    TP = 0.0
```

```
                  ENDIF
                  IF ( PERIOD - TA .LE. DELTAT )
      1           .   TA = 0.0
                  ENDIF
                  IF ( PERIOD - TS .LE. DELTAT )
      1           .   TS = 0.0
                  ENDIF
                  IF ( TP .EQ. 0.0 )
      1           .   IF (TS .GE. TP .AND. TS .LE. TA)
      2           .   .   M = 1                           $    T ON INCREASING RADIUS
      1           .   ELSE
      2           .   .   M = 2                           $    T ON DECREASING RADIUS
      1           .   ENDIF
                  ORIF ( TA .EQ. 0.0 )
      1           .   IF (TS .GT. TA .AND. TS .LT. TP)
      2           .   .   M = 2
      1           .   ELSE
      2           .   .   M = 1
      1           .   ENDIF
                  ORIF ( TP .GT. TA )            $    T-ZERO ON INCREASING RADIUS
      1           .   IF ( TS .GT. TA .AND. TS .LT. TP )
      2           .   .   M = 2                           $    T ON DECREASING RADIUS
      1           .   ELSE
      2           .   .   M = 1                           $    T ON INCREASING RADIUS
      1           .   ENDIF
                  ELSE          $ T-ZERO ON DECREASING RADIUS
      1           .   IF (TS .GE. TP .AND. TS .LE. TA)
      2           .   .   M = 1                           $    T ON INCREASING RADIUS
      1           .   ELSE
      2           .   .   M = 2                           $    T ON DECREASING RADIUS
      1           .   ENDIF
                  ENDIF
           C
                  T = ORBIT(7) + ORBTIM( M, R, ORBIT(9), ORBIT(6), ORBIT(8) )
           C         ORBTIM RETURNS TIME SINCE PERIGEE PASSAGE
           C
                  T = AMOD(T,PERIOD)
                  IF ( PERIOD - T .LE. DELTAT )
      1           .   T = 0.0
                  ENDIF
                  ASSERT ( TS .GE. T - DELTAT )
                  FAIL ( TIME )
                  ASSERT ( TS .LE. T + DELTAT )
                  FAIL ( TIME )
           C
           C         VERIFY THAT THE MODE AND VALUE INPUTS ARE SATISFIED.
           C
                  CASE OF ( MODE )
      1    C      *
                  CASE ( 0 )                     $    VALUE IS TRUE ANOMALY
      1           .   Q = (ORBIT(5) / R - 1.0 ) / ORBIT(6)
      1           .   IF ( ABS(Q) .GT. 1.0 .AND. ABS(Q) .LT. 1.0 + EPS )
                 1.       Q = SIGN(1.0, Q)
      1           .   Q = ACOS ( Q )
      1    C      *       SEE NOTES PAGE 75
      1           .   IF ( M .EQ. 2 ) Q = TWOPI - Q
      1           .   IF ( ABS(TWOPI-Q) .LE. DTHETA )
      2           .   .   Q = TWOPI
      1           .   ENDIF
      1           .   IF ( ABS(TWOPI-VALUE) .LE. DTHETA )
```

```
2              .   .   VALUE = TWOPI
1              .   ENDIF
1              .   ASSERT (AMOD(Q,TWOPI) .GE. AMOD(VALUE,TWOPI)-DTHETA)
1              .   FAIL ( PRINT Q VALUE )
1              .   ASSERT (AMOD(Q,TWOPI) .LE. AMOD(VALUE,TWOPI)+DTHETA)
1              .   FAIL ( PRINT Q VALUE )
1     C        *
               CASE ( 1, 2 )                $   VALUE IS RADIUS
1              .   IF ( (TS .NE. TA) .AND. (TS .NE. TP) )
2              .   .   ASSERT ( MODE .EQ. M )
2              .   .   FAIL ( MODE ERROR )
1              .   END IF
1              .   ASSERT ( R .GE. VALUE - EPS )
1              .   FAIL ( R VALUE )
1              .   ASSERT ( R .LE. VALUE + EPS )
1              .   FAIL ( R VALUE )
1     C        *
               CASE ( 3 )                   $   VALUE IS TIME
1     C        *
               CASE ( 4, 5 )                $   VALUE IS ALTITUDE
1              .   IF ( (TS .NE. TA) .AND. (TS .NE. TP) )
2              .   .   ASSERT ( MODE .EQ. M+3 )
2              .   .   FAIL ( MODE ERROR )
1              .   END IF
1              .   ASSERT ( R - RBODY .GE. VALUE - EPS )
1              .   FAIL ( R RBODY VALUE )
1              .   ASSERT ( R - RBODY .LE. VALUE + EPS )
1              .   FAIL ( R RBODY VALUE )
1     C        *
               CASE ELSE
1              .   ASSERT ( .FALSE. )
               END CASE
      C
      C
               BLOCK ( SEMI-MAJOR AXIS )
1              .   WRITE(6,1000) R, V, GCON, A, ORBIT(9)
1      1000    .   FORMAT (*0R=* G24.18, 5X *V=* G24.18, 5X *GCON=* G24.18 /
            1,       * A=* G24.18, 5X *ORBIT(9)=* G24.18)
               END BLOCK
      C
               BLOCK ( TIME )
1              .   WRITE(6,1001) TS, T, PERIOD
1      1001    .   FORMAT(*0TS=* G24.18, 5X *T=* G24.18, 5X *PERIOD=* G24.18)
               END BLOCK
      C
               BLOCK ( PRINT Q VALUE )
1              .   WRITE(6,1003) Q, VALUE
1      1003    .   FORMAT (*0Q=* G24.18, 5X *VALUE=* G24.18)
               END BLOCK
      C
               BLOCK ( MODE ERROR )
1              .   WRITE (6, 1004)  MODE, M, TP, TA, TS, STATE(5)
1      1004    .   FORMAT (*0MODE=* I5, 5X *M=* I5 / * TP=* G24.18, 5X *TA=* G24.18,
            1.       *TS=* G24.18 / * VR=* G24.18 )
               END BLOCK
      C
               BLOCK ( R VALUE )
1              .   WRITE (6,1005) R, VALUE
1      1005    .   FORMAT(*0R=*, G24.18, 5X, *VALUE=*, G24.18)
               END BLOCK
      C
               BLOCK ( R RBODY VALUE )
1              .   WRITE(6,1006) R-RBODY, VALUE, R, RBODY
1      1006    .   FORMAT(*0R-RBODY=*, G24.18, 5X, *VALUE=*, G24.18 /
            1.             * R=*, G24.18, 5X *RBODY=*, G24.18 )
               END BLOCK
               RETURN
               END
```

SEQ NEST SOURCE

```
C
C
C/     LIST,ALL
       SUBROUTINE ORBP (MODE,VALU,ORBEL,STATE)
CASON
CMOON ORBP
CXCOM $
C
C      SOURCE DATE 69.1231  SET UP ACCEL COMPONENTS
C      SOURCE DATE 69.0709    REMOVE CALL OF RITEF
C      SOURCE DATE 69.0321      REVISE TEST FOR APOGEE/PERIGEE
C      SOURCE DATE 68.0614  CALL RITEF, NOT CRIO / CHECK FOR ILLEGAL RADI
*      SOURCE DATE 68.0209        CONVERT TO CDC 6400
C      SOURCE DATE 67.1121     CORRECT POTENTIAL OVERFLOW ERROR
C      SOURCE DATE 67.0811    ADD OPTIONAL MODES 4,5
C      SOURCE DATE 67.0714    CALL TRAICERR IF ERROR CONDITION
C      SOURCE DATE  66.0920     USE ECC ANOMALY AS ITERATION VARIABLE
C      SOURCE DATE 66.0601    USE RADIUS/ANGLE/TIME AS ITERATION VARIABLE
C
C          RETURNS STATE VECTOR OF A POINT ON A KEPLER ORBIT
C
C          MODE   -  SELECTS SPECIFICATION OF POINT IN ORBIT
C                    0  -  VALUE IS TRUE ANOMALY
C                    1  -  VALUE IS RADIUS (RADIUS INCREASING IN TIME)
C                    2  -  VALUE IS RADIUS (RADIUS DECREASING IN TIME)
C                    3  -  VALUE IS TIME AT WHICH POINT IS REACHED
C                    4  -  VALUE IS ALTITUDE (INCREASING IN TIME)
C                    5  -  VALUE IS ALTITUDE (DECREASING IN TIME)
C          VALUE  -  PARAMETER VALUE SPECIFYING POINT IN ORBIT
C          ORBEL  -  ORBITAL ELEMENT VECTOR
C          STATE  -  STATE VECTOR AT SPECIFIED POINT
C
C          WRITTEN 12/7/64
C
C      CONCON
       COMMON /CONCON/
     1   PI,       SRD,       SLV,       SMF,       SKP,       RBODY,
     2   GACC,     GCON,      WBODY,     RHOZRO,    TWOPI,     HAFPI
CCOMPKG,CONCON
       DIMENSION ORBEL(10),STATE(10),AXES(10,3),SP(10),OE(10)
       EQUIVALENCE (SP(1),T),(SP(2),R),(SP(5),VR),(SP(6),VQ)
       EQUIVALENCE (OE(5),P),(OE(6),E),(OE(7),TP),(OE(8),PP),(OE(9),A)
       LOGICAL   INPCHK                $   VERIFIES INITIAL CONDITIONS
       LOGICAL   OUTCHK                $   VERIFIES FINAL CONDITIONS
       REAL   ORB(10)
       EQUIVALENCE ( ORB(1), OE(1) )
       DATA     EPS        / 1E-6 /
       DATA     DELTAT     / 1E-2 /      $ ABSOLUTE TIME TOLERANCE
       DATA     SP      / 10*0. /
       DATA     AXES    / 30*0. /
       DATA     EMISS   / 1.E-7 /
C

       RELERR(X,Y) = ABS(ABS(X)-ABS(Y))/AMAX1(ABS(X),ABS(Y))

       INITIAL ( INPCHK( MODE, VALU, ORBEL, STATE ) )
C
       MODD=MODE
       VALUE=VALU
       IF(MODD.LT.4) GO TO 2
```

SEQ NEST SOURCE

```
      C A100
            VALUE=VALU-RBODY
            MOOD=MOOD-3
       2    CONTINUE
            ASSERT ( MOOD .GE. 0 )
            ASSERT ( MOOD .LE. 3 )
            KODE = MOOD + 1
            CALL XMIT(8,ORBEL(2),OE(2))
            ASSERT ( OE(2) .EQ. ORBEL(2) )
            FAIL ( FIX 2 )
            ASSERT ( OE(3) .EQ. ORBEL(3) )
            FAIL ( FIX 3 )
            ASSERT ( OE(4) .EQ. ORBEL(4) )
            FAIL ( FIX 4 )
            ASSERT ( OE(5) .EQ. ORBEL(5) )
            FAIL ( FIX 5 )
            ASSERT ( OE(6) .EQ. ORBEL(6) )
            FAIL ( FIX 6 )
            ASSERT ( OE(7) .EQ. ORBEL(7) )
            FAIL ( FIX 7 )
            ASSERT ( OE(8) .EQ. ORBEL(8) )
            FAIL ( FIX 8 )
            ASSERT ( OE(9) .EQ. ORBEL(9) )
            FAIL ( FIX 9 )
            ASSERT ( E .GT. 0.0 )
            ASSERT ( E .LT. 1.0 )
            IF(E.GE.1.) GO TO 250
            ASSERT ( KODE .GE. 1 )
            FAIL ( FIX KODE )
            ASSERT ( KODE .LE. 4 )
            FAIL ( FIX KODE )
            GO TO (10,20,20,30),KODE
      C
      C     VALUE IS ANGLE
      C
         10 CONTINUE
            ASSERT ( VALUE .GE. 0.0 )
            ASSERT ( VALUE .LE. TWOPI )
            INVOKE ( ELLIPSE )
      C
            T = ORBTIM( MOOD, VALUE, A, E, PP ) + TP
            ASSERT ( T .GE. 0.0 )
            Q=VALUE
            R=P/(1.+E*COS(Q))
            ASSERT (R .GE. A*(1.0-E) - EPS)
            ASSERT (R .LE. A*(1.0+E) + EPS)
            GO TO 200
      C
      C     VALUE IS RADIUS
      C
         20 CONTINUE
            INVOKE ( ELLIPSE )
            ASSERT ( VALUE .GE. ( A * ( 1.0 - E ) ) )
            ASSERT ( VALUE .LE. ( A * ( 1.0 + E ) ) )
      C
            IF ( VALUE .LT. ( A * ( 1.0 - E ) ) ) GOTO 24
            IF ( VALUE .GT. ( A * ( 1.0 + E ) ) ) GOTO 26
         22 T=ORBTIM(MOOD,VALUE,A,E,PP)+TP
            ASSERT ( T .GE. 0.0 )
            R=VALUE
```

SEQ NEST SOURCE

```
      C          THIS CHANGE CORRECTS AN ERROR WHICH OCCURS WHEN THE
      C          RADIUS IS EQUAL TO THE PERIGEE DISTANCE.  IN THIS
      C          CASE, ERRORS ACCUMULATED DURING THE COMPUTATION CAUSE
      C          THE VALUE OF THE ARGUMENT OF THE ARC COSINE FUNCTION
      C          TO BE SLIGHTLY LARGER THAN ONE.
               QPRIME = ADIV(P-R, R*E)
               IF ( ABS(QPRIME) .GT. 1.0 .AND. ABS(QPRIME) .LT. 1.0 + EPS )
              1   QPRIME = SIGN(1.0, QPRIME)
               ASSERT (QPRIME .GE. -1.0)
               FAIL ( PRINT QPRIME )
               ASSERT ( QPRIME .LE. 1.0 )
               FAIL ( PRINT QPRIME )
               Q = ACOS (QPRIME)
               IF(MODE.EQ.3) Q=TWOPI - Q
      C                         (RADIUS IS DECREASING)
               GO TO 200
      C
      C        BELOW PERIGEE
            24 T=TP
               R=A*(1.-E)
               Q=0.
               WRITE ( 6, 280 )
           280 FORMAT (51H0CRBP HAS REPLACED IMPOSSIBLE RADIUS WITH PERIGEE - )
               GOTO 200
      C        ABOVE APOGEE
            26 T=TP+PI*PP
               R=A*(1.+E)
               Q=PI
               WRITE(6,281)
           281   FORMAT (50H0ORBP HAS REPLACED IMPOSSIBLE RADIUS WITH APOGEE - )
               GO TO 200
      C
      C        VALUE IS TIME
      C
            30 CONTINUE
               INVOKE ( ELLIPSE )
               ASSERT ( VALUE .GE. 0.0 )
               ASSERT ( VALUE .LE. (PP*TWOPI + EPS) )
               FAIL ( TIME MAX )
      C
               FM = ( VALUE - TP ) / PP
               ASSERT ( ABS(FM) .LE. TWOPI + EPS )
               FAIL ( FM MAX )

               T=VALUE
               EA1=FM
               NTRY=-1
           41  CONTINUE
               MTRY = NTRY
               NTRY = NTRY + 1
               IF(NTRY.EQ.20) GO TO 250
               EA = FM + E * SIN(EA1)
               IF(ABS(EA1-EA) .LE. EMISS) GO TO 42
               IF(MOD(NTRY,2) .EQ. 1) 45,46
            45 CONTINUE
      C
               ASSERT ( ABS(EA1 - EA2 ) .GT. 0.0 )
               FAIL ( SMALL DIVISOR )
               ASSERT ( ABS( (EA - EA1) / (EA1 - EA2) ) .LT. 1.0 )
      C
```

SEQ NEST SOURCE

```
                ASSERT ( ABS(EA+EA2-2.0*EA1) .GT. 0.0 )
                FAIL ( SMALL DIVISOR )
                EA1=EA2-(EA1-EA2)**2/(EA+EA2-2.*EA1)
                ASSERT ( MTRY .LT. NTRY )
                GO TO 41
             46 EA2=EA1
                EA1=EA
                ASSERT (MTRY .LT. NTRY )
                GO TO 41
             42 X2=1.+COS(EA)
                ASSERT ( X2 .GE. 0.0 )
                ASSERT ( X2 .LE. 2.0 )
                Q=PI
                IF(X2.LE.0.) GO TO 48
                ASSERT ( X2 .GT. 0.0 )
                X1=SQRT((1.+E)/(1.-E))*SIN(EA)
                Q=2.*ATAN2(X1,X2)
          48    CONTINUE
      C            KEPLER'S EQUATION
                ASSERT ( FM .GE. EA - E * SIN(EA) - EPS )
                ASSERT (FM .LE. EA-E*SIN(EA) + EPS)
                R = P / ( 1.0 + E * COS(Q) )
                ASSERT ( R .GE. A * ( 1.0 - E ) - EPS )
                ASSERT ( R .LE. A * ( 1.0 + E ) + EPS )
                GO TO 200
      C
      C
      C
      C         EUREKA .... NOW SET UP ANSWER AND RETURN
            200 VQ=SQRT(GCON*P)/R
                ASSERT (VQ .LE. SQRT ( ( (1.0+E)*GCON) / ((1.0-E)*A)) + EPS)
                FAIL ( PRINT VQ GCON A E )
                ASSERT (VQ .GE. SQRT ( ( (1.0-E)*GCON) / ((1.0+E)*A)) - EPS)
                FAIL ( PRINT VQ GCON A E )
                VR=R*E*SIN(Q)*VQ/P
      C            AT PERIGEE
                ASSERT ( ABS(VR) .GE. ABS( E / (1.0 + E ) * VQ * SIN(Q) ) - EPS )
                FAIL ( PRINT VR E VQ )
      C            AT APOGEE
                ASSERT ( ABS(VR) .LE. ABS ( E / (1.0-E ) * VQ * SIN(Q) ) + EPS )
                FAIL ( PRINT VR E VQ )
                OE(4)=OE(4)+Q
                ASSERT ( OE(4) .GE. -PI )
                FAIL ( PRINT OE4 )
                ASSERT (OE(4) .LE. TWOPI + TWOPI)
                FAIL ( PRINT OE4 )
      C

      C
                CALL EULANG(-1,AXES,OE,0)
      C
                NDERIV = 0
                INVOKE ( OKAXES )
      C
                CALL TRNSFM(STATE,AXES,SP,-1,1)
                CALL GRAV(STATE,STATE(8))
      C
                FINAL ( OUTCHK( MODE, VALU, ORBEL, STATE ) )
      C
                RETURN
```

```
SEQ NEST SOURCE

        C
        C
        C       ERROR MESSAGE
        C
            250 CONTINUE
                WRITE(6,251)
            251 FORMAT( 38HOCRBP HAS FAILED TO REACH A SOLUTION - )
                WRITE(6,1) OE
              1 FORMAT(* ORBITAL ELEMENTS ARE*/(5G20.8))

                BLOCK  ( ELLIPSE )
   1    C       *     VERIFIES THAT ORBITAL VECTOR @ORB@ IS AN ELLIPTIC ORBIT
   1            .   ASSERT ( ORB(2) .GE. 0.0 )
   1            .   ASSERT ( ORB(2) .LE. TWOPI + EPS )
   1            .   ASSERT ( ORB(3) .GE. 0.0 )
   1            .   ASSERT ( ORB(3) .LE. PI )
   1            .   ASSERT(ORB(4) .GE. 0.0)
   1            .   ASSERT (ORB(4) .LE. TWOPI + EPS )
   1            .   FAIL ( PRINT PERIGEE ANGLE )
   1            .   ASSERT ( ORB(5) .GE. ABS ( ORB(9) * ( 1.0 - ORB(6) ** 2 ) - EPS ))
   1            .   ASSERT ( ORB(5) .LE. ABS ( ORB(9) * ( 1.0 - ORB(6) ** 2 ) + EPS ))
   1            .   ASSERT ( ( 1.0 - ORB(6)**2 ) .LT. 1.0 )
   1            .   ASSERT ( ORB(8) .GE. ORB(9) * SQRT ( ORB(9) / GCON ) - EPS )
   1            .   ASSERT ( ORB(8) .LE. ORB(9) * SQRT ( ORB(9) / GCON ) + EPS )
   1            .   ASSERT ( RELERR(ORB(9)**3, GCON*ORB(8)**2) .LE. EPS)
   1            .   ASSERT ( ORB(6) .GT. 0.0  .AND.  ORB(6) .LT. 1.0 )
                END BLOCK

                BLOCK  ( CKAXES )
   1    C       *     VERIFIES DIRECTION COSINE ARRAYS
   1            .   INITIAL ( NDERIV .GE. 0  .AND.  NDERIV .LE. 2 )
   1            .   N = 3 * NDERIV + 4
   1            .   DO ( I = 2, N )
   2    C       *   *
   2            .   .   DO ( J = 1, 3 )
   3            .   .   .   ASSERT ( ABS( AXES(I,J) ) .LE. 1.0 )
   2            .   .   END DO
   2            .   .   DO ( J = 2, N )
   3            .   .   .   DOT = 0.0
   3            .   .   .   DO ( K = 1, 3 )
   4            .   .   .   .   DOT = DOT + AXES(I,K) * AXES(J,K)
   3            .   .   .   END DO
   3            .   .   .   IF ( I .EQ. J )
   4            .   .   .   .   ASSERT ( ABS( DOT - 1.0 ) .LT. EPS )
   3            .   .   .   ELSE
   4            .   .   .   .   ASSERT ( ABS( DOT ) .LT. EPS )
   3            .   .   .   END IF
   2            .   .   END DO
   1            .   END DO
                END BLOCK
        C
                BLOCK ( FIX 2 )
   1            .   OE(2) = ORBEL(2)
                END BLOCK
        C
                BLOCK ( FIX 3 )
   1            .   OE(3) = ORBEL(3)
                END BLOCK
        C
                BLOCK ( FIX 4 )
```

```
SEQ NEST SOURCE

     1          .    OE(4) = CRBEL(4)
                END BLOCK
          C
                BLOCK ( FIX 5 )
     1          .    OE(5) = CRBEL(5)
                END BLOCK
          C
                BLOCK ( FIX 6 )
     1          .    OE(6) = CRBEL(6)
                END BLOCK
          C
                BLOCK ( FIX 7 )
     1          .    OE(7) = CRBEL(7)
                END BLOCK
          C
                BLOCK ( FIX 8 )
     1          .    OE(8) = CRBEL(8)
                END BLOCK
          C
                BLOCK ( FIX 9 )
     1          .    OE(9) = CRBEL(9)
                END BLOCK
          C
                BLOCK ( PRINT PERIGEE ANGLE )
     1          .    WRITE ( 6, 1000 ) ORB(4)
     1    1000  .    FORMAT (*ARGUMENT OF THE PERIGEE = *, G24.18 )
                END BLOCK
          C
                BLOCK ( FIX KODE )
     1          .    IF ( MODE .GE. 0 .AND. MODE .LE. 3 )
     2          .    .    KODE = MODE + 1
     1          .    ORIF ( MODE .EQ. 4 .OR. MODE .EQ. 5 )
     2          .    .    KODE = MODE - 2
     1          .    ELSE
     2          .    .    KODE = 1
     1          .    END IF
                END BLOCK
          C
                BLOCK ( SMALL DIVISOR )
     1          .    GOTO 42
                END BLOCK
          C
                BLOCK ( PRINT VQ GCON A E )
     1          .    WRITE (6,1001) VQ, GCON, A, E
     1    1001  .    FORMAT ( * VQ=* G24.18 / * GCON=* G24.18 / * A=* G24.18 /
              1.               * E=* G24.18 )
                END BLOCK
          C
                BLOCK ( PRINT QPRIME )
     1          .    WRITE (6,1002) QPRIME
     1    1002  .    FORMAT(* QPRIME=* G24.18)
     1          .    QPRIME = SIGN (1.0, QPRIME )
                END BLOCK
          C
                BLOCK ( PRINT VR E VQ )
     1          .    WRITE(6,1003) VR, E, VQ, Q
     1    1003  .    FORMAT (*0VR=* G24.18, 5X, *E=* G24.18, 5X *VQ=* G24.18 / * Q=*
              1.        G24.18)
                END BLOCK
          C
```

```
SEQ NEST SOURCE

                     BLOCK ( PRINT CE4 )
        1              .   WRITE(6,1004) QE(4), Q
        1       1004 .   FORMAT(*QOE(4)=* G24.18, 5X *Q=* G24.18)
                     END BLOCK

                     BLOCK ( TIME MAX )
        1              .   WRITE ( 6, 1005 )  VALUE, PP*TWOPI
        1       1005 .   FORMAT (* VALUE= * G24.18, 5X,10HPP*TWOPI=  ,G24.18 )
                     END BLOCK

                     BLOCK ( FM MAX )
        1              .   WRITE ( 6, 1006 ) FM, TWOPI
        1       1006 .   FORMAT ( * FM = *, G24.18, 5X, * TWOPI= * G24.18 )
                     END BLOCK

                     BLOCK ( FM )
        1              .   WRITE(6,1007) FM, VALUE, TP, PP
        1       1007 .   FORMAT(*OFM=*, G24.18 / * VALUE=*, G24.18,5X, *TP=*, G24.18, 5X,
                    1.            *PP=*, G24.18)
                     END BLOCK
              C
                     RETURN
                     END
```

****

APPENDIX B

Chronological List of Papers Submitted

The following collection of papers and reports was supported by AFOSR contract number F49620-79-C-0115.

1. D. M. Andrews, Using Assertions for Adaptive Testing of Software, presented at the International Federation of Information Processing Society Working Conference, September 26-29, 1979, London, England.

2. D. Andrews and J. Benson, Using Executable Assertions for Testing, presented at the 13th Annual Asilomar Conference on Circuits, Systems and Devices, November 6, 1979, Pacific Grove, California.

3. D. Andrews and J. Benson, An Automated Program Testing Methodology and Its Implementation, submitted to the 10th International Symposium on Fault-Tolerant Computing, October 1-3, 1980, Kyoto, Japan.

4. D. Andrews and J. Benson, Adaptive Search Techniques Applied to Software Testing, Final Report, General Research Corporation CR-1-925, February, 1980.

5. J. Benson, A Preliminary Experiment in Automated Software Testing, General Research Corporation TM-2308, February, 1980.

6. D. M. Andrews and J. P. Benson, "Using Assertions to Test Computer Programs Automatically", to be submitted to the IEEE Transactions on Software Engineering.

APPENDIX C

Personnel Associated with the Project

The following persons participated in the research and experiments:

1. Dorothy M. Andrews, MSEE, University of California, Santa Barbara.

2. Jeoffrey P. Benson, PhD., University of California, Santa Barbara.

3. Nancy B. Brooks, MS, University of Illinois.

4. Reginald N. Meeson, MSEE, University of California, Santa Barbara.

5. Dennis W. Cooper, MSEE, Stanford University.